Mutation analysis applied to security tests

Master of research in computer science report

Author: Tejeddine Mouelhi
Supervisors: Yves Le Traon, Benoit Baudry

ENST Bretagne - June 2007

Acknowledgements

I would like to gratefully acknowledge the support of my advisors Yves le Traon and Benoit Baudry. I am very grateful to them for their helpful advices.

Table of contents

List of figures

List of tables

1. Introduction

According to the CERT [1], the number of reported vulnerabilities in 2006 was 8064 (5990 were reported in 2005, a 36% increase). As a consequence, and overall justification for this research work, the international community needs new approaches to deal with these security issues. One of the open challenges is to provide web application developers with pragmatic techniques to check that the security aspects are implemented according to the security requirements. No trust can be built in security aspects if the developers cannot even check that these security aspects have been correctly implemented. The current report studies how security can become a target for testing and which criteria can be applied to have some trust in the security mechanism implementation. The goal of this work is to offer a first trustable step in a design for security process: it doesn't aim to generate attacks but "only" to test that a security mechanism is consistent with what it is supposed to do.

Currently, an important effort is dedicated to software testing in development cycles, which varies between 35 to 55% of the overall effort for producing an application (from requirements elicitation to system delivery to final clients). Except ad-hoc penetration testing techniques, very few works have been dedicated to the issue of testing security in the same way functional aspects are tested. The first difficulty is to estimate test cases quality and specify the faults specific to the domain (e.g. security flaws). Among the techniques for estimating test quality, *mutation analysis* is the only one which builds confidence in test cases in relation to an explicit fault model. .Mutation analysis consists of creating faulty versions of a program and checking whether the test cases are efficient – or not - to detect these injected faults. This technique is usually used for three main purposes: estimating the efficiency of a testing technique, measuring the robustness/fault tolerance of a system and as an objective to generate tests (in that case, it is called *mutation testing*). In this report, we adapt mutation and perform studies for two different security aspects of a 3-tiers architectures, namely security policy (access control) and SQL injection attacks. Mutation analysis is used :

- as a way to compare functional tests and tests generated for checking the correctness of a security mechanisms (security policy testing),
- as an objective for testing security policies,
- to check the robustness of security mechanism in the context of SQL injection attacks (SQLIA).

The first part of this report will focus on the adaptation of mutation analysis for testing access control based security policies. The considered access control model is OrBAC, an advanced model developed by the SERES team in the RSM lab. of the ENST-Bretagne engineering school. OrBAC is a security model for defining security policy access rules and the implementation of these rules has to be tested for a given deployed system. In this part of the report, the use of mutation analysis is two-fold. On one hand, it serves for experimental purposes as a technique to estimate the efficiency of testing techniques for security. On the other hand, the feasibility of the use of mutation for generating tests is studied (mutation testing). The OrBAC security model will be presented and the mutation analysis adaptation will be introduced using a simple case study. This study, and the first results it provides, leads to a ranking of the mutation operators (types of injected faults) and gives a comparison of security tests and functional tests. In addition, testing strategies are evaluated using the results of these experiments. The results and the study conducted were submitted to the ISSRE conference. The article title is "Testing security policy: going beyond functional testing". In it, we present the results concerning the relation between functional and security tests and the comparison between the proposed strategies. In addition, we have submitted another article to Mutation 2007 conference. The article title is "Mutation analysis for security tests

qualification". It presents in details the implementation of the mutation analysis, the mutation operators and the ranking of the operators.
In the second part, we apply the mutation analysis approach in the context of SQL injection attacks (SQLIA) which are among the most important attacks performed against web applications. In the literature, several security mechanisms were proposed to deal with SQLIA. We present the main ones and propose a testing methodology based on fault injection in order to estimate the robustness of each of these security mechanisms. This study - rather qualitative due to the duration of the internship –highlights the potential weaknesses of each security mechanism and leads to appropriate improvements and countermeasures.

2. Background and definitions

Testing that a system is correct with respect to security is known as a hard task [2]. Among the numerous identified difficulties we can first point that security issues are handled at many different places in a system (network, hardware, server and client). Moreover, since specifying the expected security qualities is complex, it is very difficult to express the expected result when building test cases for security.
In this report, we restrict our study to 3-tier applications, even if the proposed solutions can be applied to other contexts (e.g. n-tiers architectures). As shown in Fig 1, there are 3 main parts in the application architecture; the database, the application server and the interface. Mutation analysis will be adapted to two different security testing contexts.
Firstly, we focus on OrBAC based access control model use for the 3-tier application. Security test cases are generated to check the correctness of the implementation with respect to those security rules. Today there is no systematic way to derive test cases from a security policy and no test adequacy criteria to assess the quality of test cases for security. Thus, once test cases have been produced, it is necessary to estimate their quality in terms of ability to detect security *flaws* in the implementation.
In order to evaluate the quality of test cases for security, we adapt mutation analysis and introduce new mutation operators that correspond to fault models for access control security policies. We introduce a running example used for illustration. This example is based on a library management system.
Secondly, mutation analysis will be applied to evaluate SQL injection countermeasures. The strategy used in this part consists on injecting faults into security mechanism environment in order to perturb the behaviour of the application.

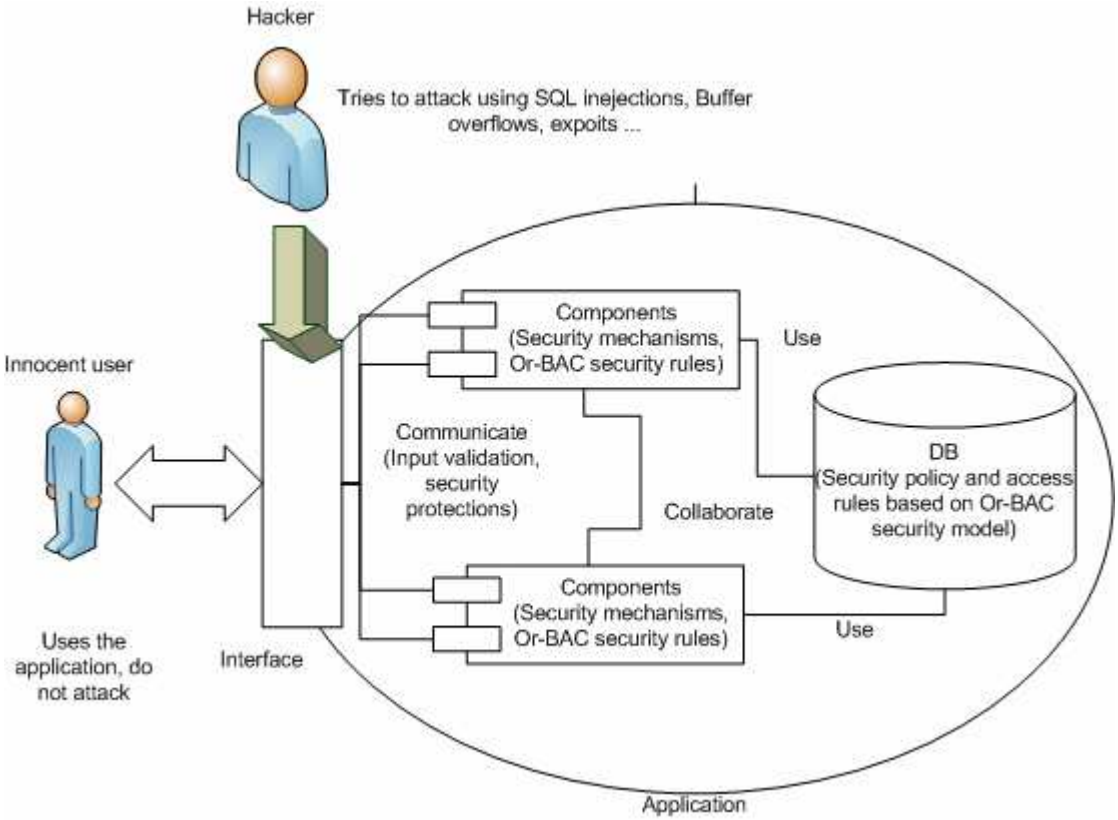

Fig 1. General context

In this section we present the mutation analysis technique. Then we present related works concerning the application of the mutation analysis in the context of security testing.

2.1. Mutation analysis

Mutation analysis technique was initially proposed by DeMillo [3] and consists in creating a set erroneous versions of the program under test that are called *mutants*. The goal for the tester is then to write a series of tests which makes it possible to distinguish the initial program from all its mutants. This technique thus makes it possible to write relevant tests.

In practice, the mutants are created by submitting the program to operators that insert simple errors* (changing the sign of a constant, replacing arithmetic operators…). Indeed, DeMillo limits the analysis to the injection of simple errors and makes the assumption that if a set of tests can detect all the simple errors, then it will be able to detect more complex errors in the initial program. If a test case applied on a mutant results in an output that is different from the output produced by the initial program, the test case has detected the mutant. The test case *killed* the mutant, in the other case the mutant is still *alive*. A mutant which is detected as erroneous by a test case is said to be killed by the test case.

Among the generated mutants some are equivalent to the initial program, i.e. no input data makes can distinguish the two programs. Fig 2 illustrates an example of equivalent mutant: (from [4]).

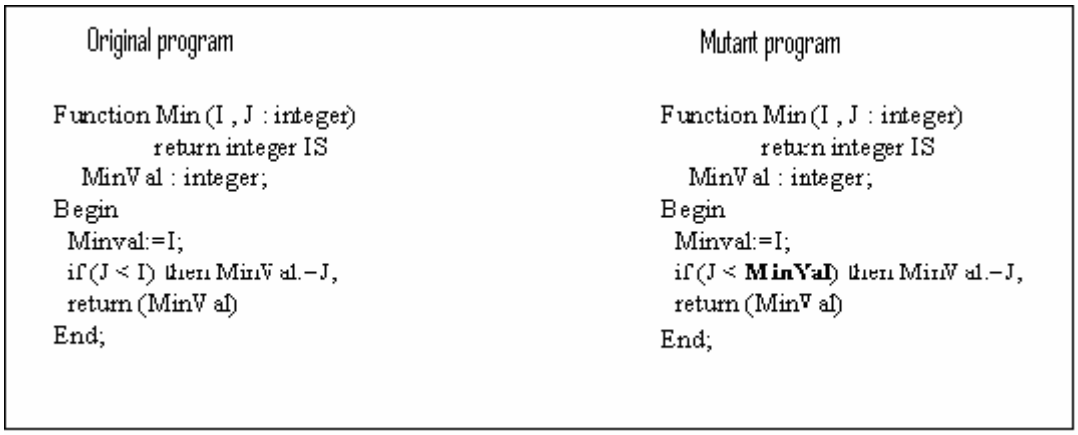

Fig 2. Example of equivalent mutant

In this example, the mutant program replaced I by MinVal in the second instruction. However, the previous instruction assigns to MinVal the value 1. We can see that variables MinVal=I having same value, the mutant program is equivalent to the initial program.

It is important to determine equivalent mutants, because they must be removed from the set of mutants to be killed. Without that there will be never a set of tests able to detect all the mutants since some will be impossible to detect, and then we will not be able to rely on our tests. The detection of equivalent mutants is generally done by hand, which increases the cost of the analysis of change. The tests cases are thus confronted to the set of non-equivalent mutants. In this report, we intend to generate faulty security policy implementations, and on constraint for the feasibility of the approach to this context is to avoid the creation of equivalent mutant (from the security policy point of view).
The objective of the mutation analysis is to estimate the efficiency of a test cases set, that is its capacity in revealing faults (the fault revealing power of a test case). It can be used to compare test criteria, test strategies and any test technique. The *mutation score* is an indicator for the quality of the tests. Let d be the number of dead mutants and m the number of mutants, the mutation score MS is: MS(t) = d/m.
The process of mutation analysis is presented in Fig 3.

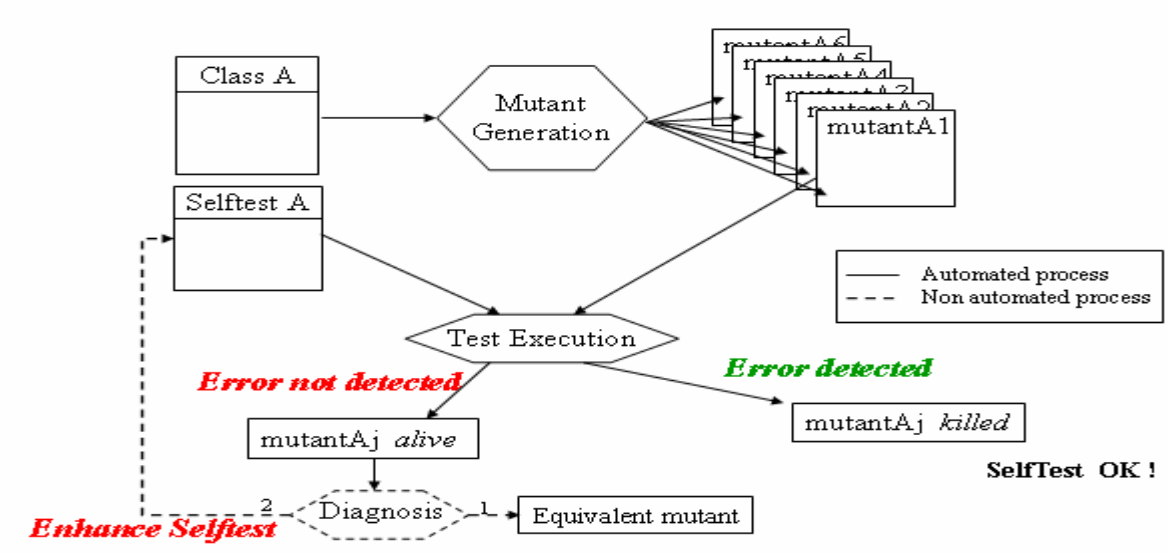

Fig 3. Mutation analysis process

2.2. Fault injection applied to security tests

To our knowledge, very few works proposed fault injection to security tests. In [5], Mathur et al. applied fault injection to the application environment. The application environment is perturbed by modifying environment variables, files or process used by the application under test. Then the application has to resist to this perturbation and must not have an insecure behaviour that may lead to security flaw. The authors applied their technique to an existing application used by Purdue university. This application is used by students and teacher. They were able to find a security hole that represents a real threat. This technique is based on fault injection and allows discovering new kinds of security holes that are possible to find with traditional techniques.
Furthermore, fault injection was applied in another way. Adaptative vulnerability analysis [6] injects faults to the application data flow and internal variables. The objective is to identify parts of application's code that have insecure behaviours when the state of the application is perturbed.
In addition, fault injection was applied to back box testing. The black box framework Fuzz [7] uses fault injection by randomly creating data input for a given application. The goal of this approach is to identify abnormal behaviour and to check that the application does not crash.

2.3. Functional testing vs. security policy testing

Testing system functions includes exercising many security mechanisms. The issue is to determine whether testing functions provide enough confidence in the security mechanisms. We would like to know how to improve this confidence by selecting test cases specific to security. Two strategies are proposed for producing security policy test cases, depending if they are built in complement of existing functional test cases or independently from them.
In this section we present the overall process to derive test cases from requirements. In this process we identify the main artefacts that must be produced for testing and propose precise definitions for the notions used in this report.
The software lifecycle may vary, but the input information is always a requirement document which includes the functional description of the system as well as many extra-functional concerns (performances, real-time, availability, technical and architectural constraints …). Among these concerns, the security aspects are often mixed with the functional ones, as it will be illustrated with the running example throughout the report. The requirement analysts have to extract these aspects and express, on one hand the uses cases and the business model and on the other hand explicit the security policy in the form of an access control model. It is composed of a set of security policy rules, each of them specifying rights and restrictions of actors on parts and resources of the system. The business model (a class diagram + simple dynamic models) focuses on the concepts which are needed to derive functional aspects, in a nominal use of the system. The security policy may introduce specific concepts, and reuse most of the concepts and functions identified in the uses cases and business model. As a result, the security policy makes reference to the business and use case models, but includes new concepts which are taken into account in the refinement process, either during design (SecureUML [8]) or at deployment/coding steps. In this section, we will detail these differences.
Figure 4 highlights the fact that both the code and the tests, which exercise this code, are produced from the requirements by independent ways. The important point is that the security policy test cases are obtained using the access control model, while the functional test cases are derived only from the uses cases (and business model). Security policy test cases are not only dependent on the security policy but also refer to the use cases and the business model.

In this report, the functional test cases (or system tests in the sense of Briand's work [9]) are generated using the approach presented in [10], based on the use cases improved with pre- and post-conditions, called contracts. The functional test cases cover all the code implementing the functions of the system. We will study how functional test cases can be reused for testing security mechanisms.

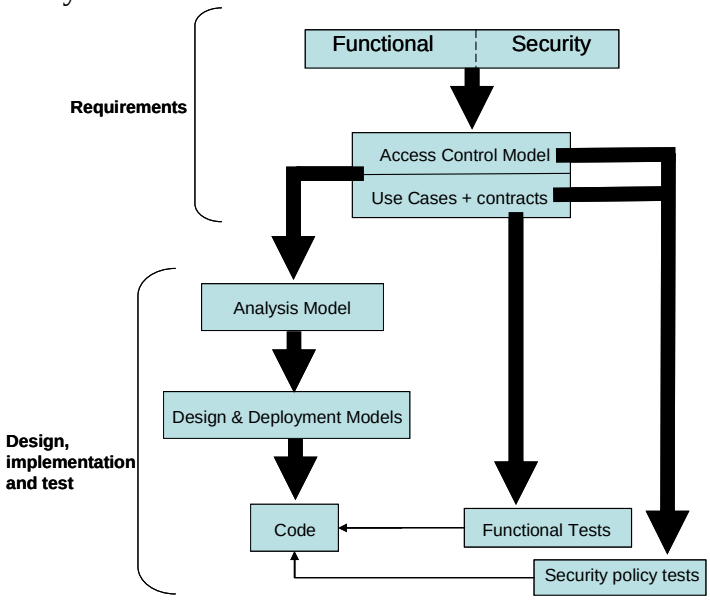

Fig 4. Process to generate security policy tests from an access control model

2.4. Definitions and differences between functional and security tests

Security Policy: *it describes the* permissions *or* prohibitions *for people to any of the resources of the system (it may apply to configure a firewall as well as to define who can access a given service or data in a database).*

The most advanced models ([11-13]) express rules that specify permissions or prohibitions that apply only to specific circumstances, called *contexts*. For instance, in the health care domain, physicians have special permissions in specific contexts, such as the context of urgency. Also, some models provide means to specify the different security policies applicable to the various parts of an organization (sub-organizations). At the end of this specification process, the security policy specifies what the permissions and prohibitions should be in the system, in function of contexts, roles and views. In this work we use OrBAC [12, 13] as a specification language to define the access control rules (a set of rules specifies a security policy). Based on the simplified requirements of this system, it is possible to derive a set of access control rules using the OrBAC model.

System/functional testing: *the activity which consists of generating and executing test cases which are produced based on the uses cases and the business models (e.g. analysis class diagram and dynamic views) of the system [9, 10] . By opposition with security tests, we call these tests* functional.

Security policy testing (SP testing): *it denotes the activity of generating and executing test cases which are derived specifically from a security policy. The objective of SP testing is to reveal as many security flaws as possible.*

Security flaw: *A security flaw is the equivalent of a fault for functional testing. It corresponds to an inconsistency between the security policy (the specification) and a security mechanism (the implementation) which is revealed at runtime.*

Test case: *In the report, we define a test case as a triplet:* intent*, input test sequence, oracle function. The* intent *may be either to test functional or security policy aspects.*

Intent of a test case: *The intent of a test case is the reason why an input test sequence and an oracle function are associated to test a specific aspect of a system. It includes at least the following information: (functional, names of the tested functions) for functional test cases or (security policy, names of the tested security rules) for SP ones.*

SP oracle function: *The oracle function for a SP test case is a specific assertion which interrogates the security mechanism. There are two different oracle functions:*

- *For permission, the oracle function checks that the service is activated.*
- *For a prohibition, the oracle checks that the service is not activated.*

In our case, services are methods in the business application, thus the oracle checks the absence or presence of a method call. It has to be noted that some data are constrained by access control rules, but the oracle function does not check any data values (in the database) because they are always accessed through a service.
The intent of the *functional* tests is not to observe that a security mechanism is executed correctly. For instance, for an actor of the system who is allowed to access a given service, the functional test intent consists of making this actor execute this service. Indirectly, the permission check mechanism has been executed, but a specific oracle function must be added to transform this functional test into a security policy test.
A security flaw may occur:

- *when an actor (person or program) of the system has (has not) access to a resource while the security policy stipulates it should not (should).*
- *when an actor (person or program) of the system has (not) access to a resource while no security policy exists for this particular actor and when this resource is the object of other security rules.*

The second point corresponds to security test cases highlighting the incompleteness in the security policy which leads to incorrect default security mechanisms. We call such test cases advanced security test cases in the following.

Examples:
Functional – a functional test case will make a borrower borrow and return a book.

Intent*: functional, test borrow and return for a unique borrower*

Oracle*: check that the book is available after it has been returned*

Security Policy – a SP test case will check that a borrower can borrow a book to the library the working days.

Intent*: security, permission for a borrower to borrow a book the working days.*

Test sequence: *create the context of a working day, make a borrower borrow a book.*

Oracle: interrogate the security mechanism and check that the permission has been computed and given to the borrower.

3. Mutation applied to OrBAC security model

Mutation analysis is a technique for evaluating the quality of test cases. We propose to apply it in the context of security penetration tests. Security tests and software tests share many aspects. We can assimilate faults to security breaches or defects in the protection mechanisms. In addition, test scenarios can be assimilated to security attacks made by a hacker. In order to apply mutation analysis to security tests we need to find suitable mutation operators.

Since there are many different solutions to implement access control rules into the business model, it is very difficult to define security fault models based on syntactic errors in the code. We could inject classical mutation faults (arithmetic, logical etc.) in security mechanisms, but the effect of these faults would be unpredictable w.r.t. a given security policy (and the verdict difficult to define).

Mutation operators are defined independently from implementation-specific details. Since the access control rules are expressed with OrBAC, we model the faults at this level of abstraction. The mutation operators are thus expressed on OrBAC syntax. On one hand, this approach has the advantage of defining faults that are actually related to the definition of security rules (prohibition instead of permission, wrong role, etc.). On the other hand, the difficulty consists in transforming mutation operators defined with OrBAC into faults in the implementation.

For this study, we assume that traceability matrices and maintained between OrBAC rules and the corresponding code blocks in the implementation. Using this information, it is easier to map mutant OrBAC rules to faults in the implementation. However, in the general case, the traceability information might not be available, or the mapping between the rules and the code might not be trivial. In that case it is not possible to automatically generate the mutants. If we still want the mutation analysis to be used, it is important to identify the most relevant subset of operators, in order to produce the minimum mutants that lead to the production of the best security test cases. This is one of the objective of the case study. It will help to select a sufficient subset of mutation operators.

In the following section, we present a set of mutation operators that can be defined for OrBAC rules. We illustrate these operators with examples. Then, in next section we run a case study to select the most relevant subset. The security mutation operators are divided in 4 categories:

- Type changing operators
- Parameter changing operators
- Hierarchy changing operators
- Adding rules operators

3.1. Security mutant operators

We identified 27 mutant operators. They are classified in 4 categories:

- Rule type mutation operators
- Rule parameter mutation operators
- Rule adding or removing mutation operators
- Hierarchy mutant operators.

We will present all of these operators. We chose to reject some of the these operators. We will explain in details the reasons. Then the selected operators will be presented in examples in order to show how they can be used and the security flaws that are simulated by the resulting policy.

3.1.1. Type changing operators

Type change operators pick a rule and modify its type. There are 3 types of rules: prohibition, permission and obligations. Therefore, there are 6 possible modifications:

1. Prohibition to permission
2. Permission to prohibition
3. Prohibition to obligation
4. Obligation to prohibition.
5. Permission to obligation
6. Obligation to permission

The first four operators simulate faults that occur during the implementation. The obtained policy allows a forbidden activity or prohibits an authorized activity.
The problem of operators 5 and 6 is that it is difficult to distinguish between obligation and permission. Test cases are not able to validate that an obligation rule is used instead of a permission one. For this reasons, we choose not to use obligation rules and only use prohibition rules. We only implement the first 2 operators (Prohibition to permission named PRP and permission to prohibition named PPR).

3.1.2. Parameter changing operators

Parameter changing operators pick a rule and change one of its parameters. As there are 5 parameters the following operators are defined:

1. Change organization
2. Change role
3. Change activity
4. Change view
5. Change context

We can not replace a rule's organization because each organization defines its own entities and we may not have the role, activity, view and context of the first organization defined for the new organization.
If we have:

```
Permission(org1,role1,activity1,view1,context1)
```

We cannot replace it with:

```
Permission(org2,role1,activity1,view1,context1)
```

We can only replace if org2 defines role1, activity1, view1 and context1 in its scope.
In addition, the notions of activity and view are tightly linked. In fact, the activity must be related to the rule view. The following example illustrates this issue:
Rule used:

```
Permission (Library, Administrator, ModifyAccount, BorrowerAccount, default)
```

Rule to use instead:

```
Permission (Library, Administrator, ReserveBook, BorrowerAccount, default)
```

`ModifyAccount` cannot be replaced by `ReserveBook` because `ReserveBook` can not be attached to the `BorrowerAccount` view. The same problem appears when we change the rule's view. In fact, replacing views or activities can be done only for activities that are independent from views (and can be applied to different views). For example, if we have the activity `Modify` that may be applied to 2 views `BorrowerAccount` and `PersonnelAccount` then we can replace `BorrowerAccount` by `PersonnelAccount`. Therefore, the relevant operators are those changing the role and the

context. These 2 operators are useful because they will simulate cases where the security policy is too permissive or too restrictive. The first one allows a user (the secretary) to do the same activity allowed for another user. The second one allows a user to perform the action under another context. In this category, we only keep two operators (Change role named RRD and change context named CRD).

3.1.3. Rule adding operators

Instead of replacing an existing rule, the adding rule operator introduces a new rule. The goal of this operator is to simulate cases where the implementation does something in addition to the requirement. This is a typical security fault and makes security faults different from functional tests. The security breaches are caused by the fact that the application behaves in unexpected way, even if it satisfies all functional requirements.
In order to obtain relevant rules, the add rule operator (named ANR) introduces rules that contain an activity and a view that were already defined by at least one rule in the initial security policy. It is important to note that this operator generates a lot of mutants.

3.1.4. Hierarchy mutant operators

OrBAC allows defining hierarchies for organisations, roles, activities, views and contexts. Then, a mutation operator can be used to change hierarchies by replacing a parameter by its parent or one of its descendants.

1. Change organization hierarchies
2. Change role hierarchies
3. Change activity hierarchies
4. Change view hierarchies
5. Change context hierarchies

Due to the reasons explained in the previous section, the first operator is not used. In addition, context and views hierarchies are not useful. In practice we do not define hierarchies for contexts and views. In fact, we insist on hierarchies for activities and roles.
The only useful operators in this category are operators 2 and 3. The hierarchy changing operators that will be retained are: ‘change rule hierarchies’ (named RPD) and ‘change activity hierarchies’ (named APD).

3.1.5. Summary of security mutation operators

This table presents the retained operators that were presented in details in previous section.

Mutation Operator	Description
PRP	Prohibition rule replaced with permission
PPR	Permission rule replaced with prohibition
RRD	Rule role is replaced with different role
CRD	Rule context is replaced with different context
RPD	Rule role (a parent) replaced with one of its descendants
APD	Rule activity replaced with one of its descendants
ANR	New rule Added

Table 1 OrBAC Mutation operators

Number of mutants

The following table shows the number of each type of mutant that can be generated. It is function of the number of entities that are in the model.

Operator name	# generated mutants
Hierarchy changing operator	:# hierarchies levels * # rules with entities (contained in hierarchies)
Addition of new rule operator	#roles * #contexts * # (activities and views)
Type changing operator	# permission + #prohibition
Parameter changing operator	# entities * # rules

Table 2 Number of generated mutants

3.2. Examples of OrBAC mutation

We illustrate the use of security mutation operators by presenting an example of security policy. The context of the study case is a hospital. We specify these entities:

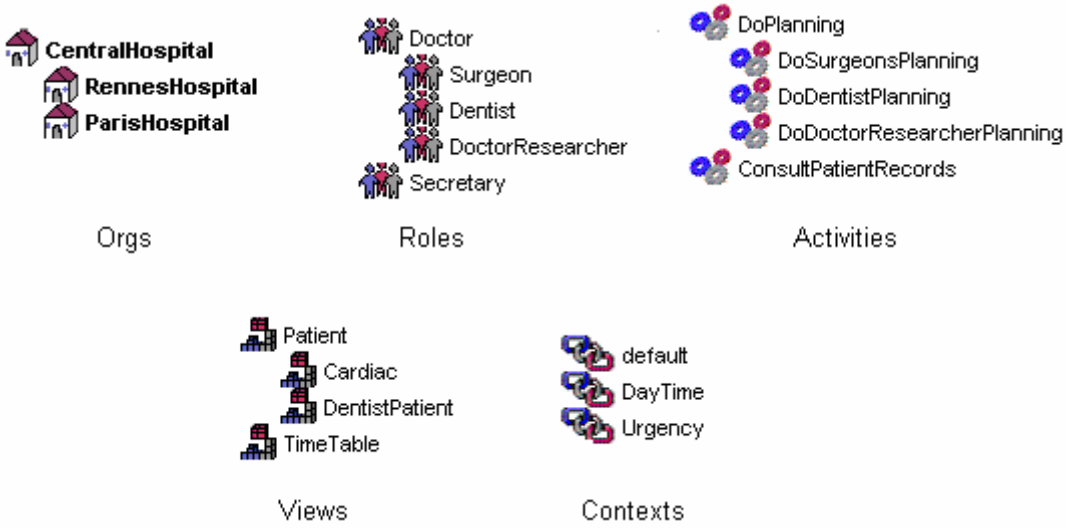

Fig 5. hospital OrBAC model entities

Some Rules are added:

We allow the secretary to do doctors planning.

permission(RennesHospital,Secretary,DoPlanning,TimeTable,Default)

permission(RennesHospital,Surgeon,DoSurgeonsPlanning,TimeTable,Default)

permission(RennesHospital,Dentist,DoDentistsPlanning,TimeTable,Default)

permission(RennesHospital,DoctorReasearcher,DoDoctorReasearcherPlanning,TimeTable, Default)

We prohibit doctors to do the planning of other roles.

prohibition(RennesHospital,Surgeon,DoDentistsPlanning,TimeTable,Default)

prohibition(RennesHospital,Surgeon,DoDoctorReasearcherPlanning,TimeTable,Default)

prohibition(RennesHospital,DoctorReasearcher,DoDentistsPlanning,TimeTable,Default)

prohibition(RennesHospital,DoctorReasearcher,DoSurgeonsPlanning,TimeTable,Default)

prohibition(RennesHospital,Dentist,DoSurgeonsPlanning,TimeTable,Default)

prohibition(RennesHospital,Dentist,DoDoctorReasearcherPlanning,TimeTable,Default)

In the following section, one mutation operator will be applied for each category to show what kinds of security violations are simulated.

3.2.1. The use of type changing operators

PPR operator:

permission(RennesHospital,Secretary,DoPlanning,TimeTable,Default)

Became

prohobition(RennesHospital,Secretary,DoPlanning,TimeTable,Default)

By using this operator, the resulting mutant simulates the violation of non repudiation security requirement.

3.2.2. The use of parameter changing operators

Applying RRD:

permission(RennesHospital,Secretary,DoPlanning,TimeTable,Default)

Became

permission(RennesHospital,Doctor,DoPlanning,TimeTable,Default)

This will allow doctors to do planning. There will be no conflicts because we defined rules or roles that inherit from “doctor” and for activities that inherit from “DoPlanning”. So these rules will be replaced by permission rule because the parent role and parent activity have priority and descendants inherits parent rules.

This policy mutant allows activities that were forbidden by the initial security policy. Authorization security requirement is violated.

3.2.3. The use of the rule adding operator

Applying ANR, the following rule is added:

permission(RennesHospital,Secretary,DoPlanning,TimeTable,Urgency)

A rule that defines the authorization granted to secretaries to do planning in context of urgency is added. The resultant mutant simulates the case when the system is too permissive.

3.2.4. The use of hierarchy mutant operators

Applying APD:

permission(RennesHospital,Secretary,DoPlanning,TimeTable,Default)

Become

permission(RennesHospital,Secretary DoDentistsPlanning,TimeTable,Default)

As the mutation operator reduces the scope of secretary activities, the resultant policy mutant simulates the violation of non repudiation security requirement.

4. Case study and mutation framework

We apply the OrBAC mutation analysis to a sample application used to manage a library. We start with a presentation the implementation of the mutation tool in MotOrBAC. Afterwards, We present the sample application OrBAC model and the architecture of the security policy deployment. Finally, we present the application and the security mutation implementation and the mutation analysis results.

4.1. Case study requirements

The purpose of the library management system (LMS) is to offer services to manage books in a public library. The books can be borrowed and returned by the users of the library on working days. When the library is closed, users can not borrow books. When a book is already borrowed, a user can make a reservation for this book. When the book is available, the user can borrow it. The LMS distinguishes three types of users: public users who can borrow 5 books for 3 weeks, students who can borrow 10 books for 3 weeks and teachers who can borrow 10 books for 2 months.
The library management system is managed by an administrator who can create, modify and remove accounts for new users. Books in the library are managed by a secretary who can order books, add them in the LMS when they are delivered. The secretary can also fix the damaged books in certain days dedicated to maintenance. When a book is damaged, it must be fixed. While it is not fixed, this book can not be borrowed but it can be reserved by a user. The director of the library has the same accesses.
The administrator and the secretary can consult all accounts of users. All users can consult the list of books in the library.

4.2. Sample application OrBAC model

We defines according to the requirements the following entities;

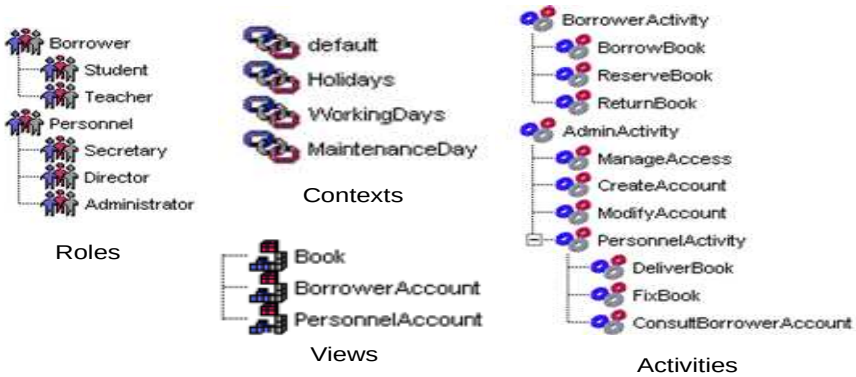

Fig 6. OrBAC entities for the LMS

The following roles are defined:

- Borrower roles : contain student and teacher
- Personnel roles : contain secretary (in charge of books), administrator (in charge of accounts, users), and the director.

Activities are classified depending on related roles. We differentiate between borrower activities and administrator activities. Therefore, there are 2 hierarchies; The borrower's activities and the personnel's activities. Fig 6 shows these roles. Note that personnel activities are sub-activities of the administrator activity because administrators have extended activities.

4.3. Security rules

The following rules are defined:

Types	Organization	Role	Activity	View	Context
Permission	rennesLibraries	Borrower	BorrowerActivity	Book	WorkingDays
Prohibition	rennesLibraries	Borrower	BorrowerActivity	Book	Holidays
Permission	rennesLibraries	Administrator	ManageAccess	PersonnelAccount	default
Permission	rennesLibraries	Administrator	CreateAccount	BorrowerAccount	default
Permission	rennesLibraries	Administrator	ModifyAccount	BorrowerAccount	default
Permission	rennesLibraries	Administrator	ConsultBorrowerAccount	BorrowerAccount	default
Permission	rennesLibraries	Administrator	FixBook	BorrowerAccount	MaintenanceDay
Permission	rennesLibraries	Administrator	DeliverBook	BorrowerAccount	MaintenanceDay
Prohibition	rennesLibraries	Secretary	ManageAccess	PersonnelAccount	default
Prohibition	rennesLibraries	Secretary	CreateAccount	BorrowerAccount	default
Prohibition	rennesLibraries	Secretary	ModifyAccount	BorrowerAccount	default
Permission	rennesLibraries	Secretary	ConsultBorrowerAccount	BorrowerAccount	default
Permission	rennesLibraries	Secretary	FixBook	Book	MaintenanceDay
Permission	rennesLibraries	Secretary	DeliverBook	Book	MaintenanceDay
Permission	rennesLibraries	Director	ConsultBorrowerAccount	BorrowerAccount	default
Prohibition	rennesLibraries	Director	ManageAccess	PersonnelAccount	default
Prohibition	rennesLibraries	Director	CreateAccount	BorrowerAccount	default
Prohibition	rennesLibraries	Director	ModifyAccount	BorrowerAccount	default
Permission	rennesLibraries	Director	FixBook	Book	MaintenanceDay
Permission	rennesLibraries	Director	DeliverBook	Book	MaintenanceDay

It is important to note that rules are derived based on the parameters hierarchy. For example the rule:

- *Permission(rennesLibraries,Borrower,BorrowerActivity,Book,WorkingDays)*

Based on hierarchies shown in Fig 6, the following rules are automatically derived:

- *Permission(rennesLibraries,Student,BorrowBook,Book,WorkingDays)*
- *Permission(rennesLibraries, Student,GiveBackBook,Book,WorkingDays)*
- *Permission(rennesLibraries, Student,ReserveBook,Book,WorkingDays)*
- *Permission(rennesLibraries,Teacher, BorrowBook,Book,WorkingDays)*
- *Permission(rennesLibraries, Teacher, GiveBackBook,Book,WorkingDays)*
- *Permission(rennesLibraries, Teacher, ReserveBook,Book,WorkingDays)*

The current security policy does not have any conflicts.

4.4. The business model and architecture

The UML class diagram is shown by Fig 7:

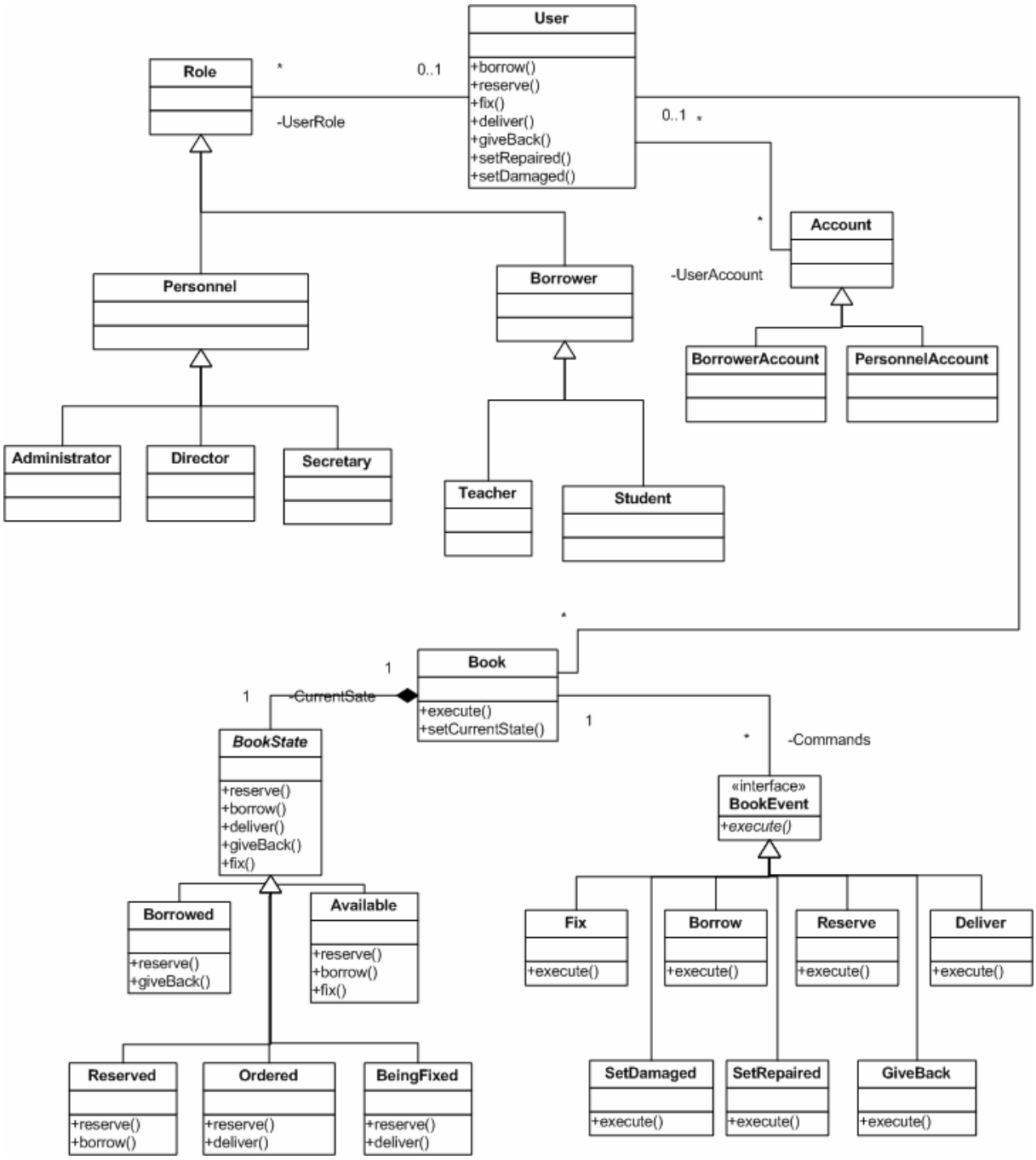

Fig 7. UML Class diagram

We managed to define an UML model that does not hard-code the security policy in it. For example we do not define a method *borrow* in the borrower class because it implements a security policy rule by construction in the model (allowing only a borrower to borrow) and therefore this rule cannot be violated or changed. In fact a generic model allows us to have a dynamic implementation of the security policy. The security policy is in charge of deciding whether an activity requested by a user under a specific context and performed under a certain view is allowed or prohibited.

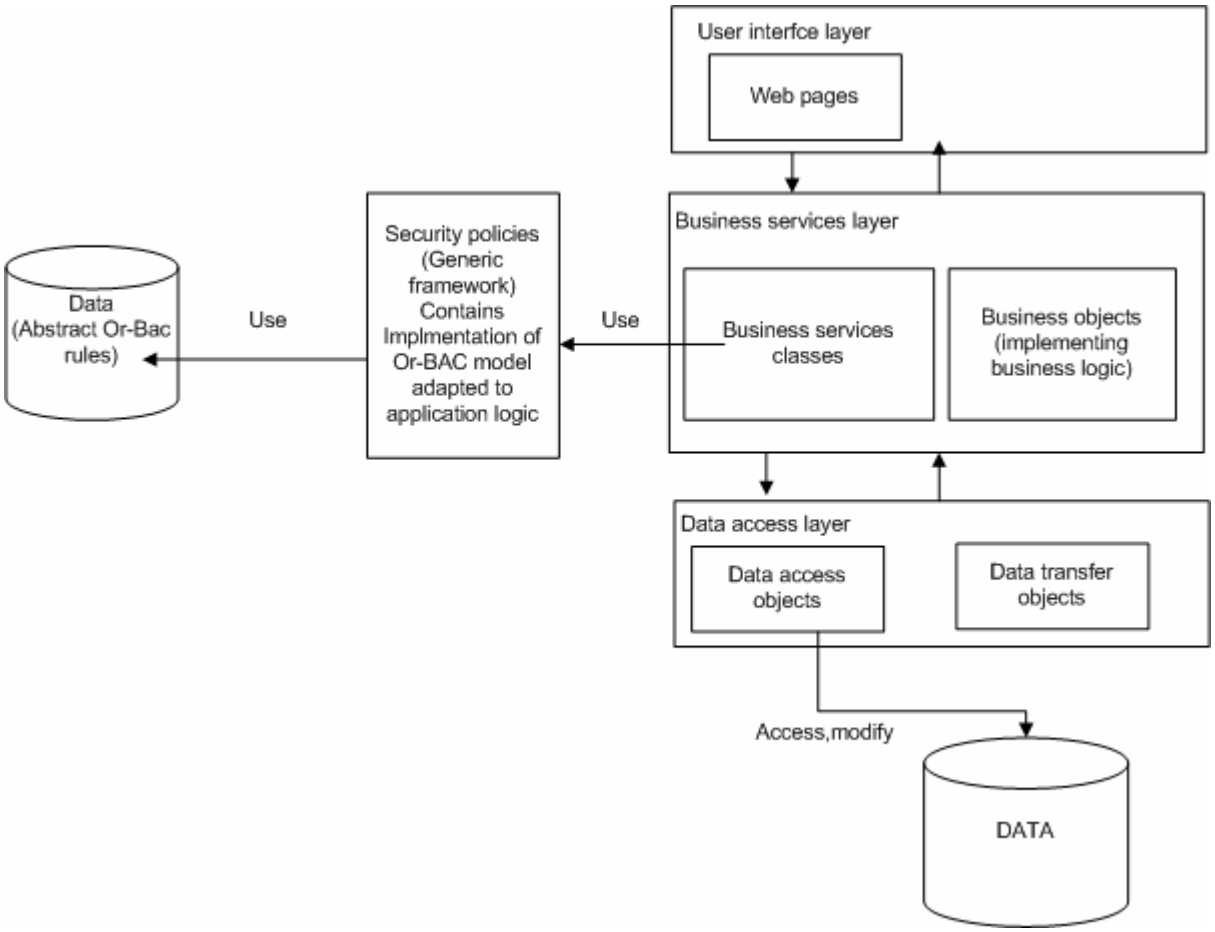

Fig 8. Application architecture

In addition to the model presented above, we use 3-tiers architecture, as shown in Fig 8. Is it important to note that all user interactions with the application are handled by service classes. These classes are implementation of facade design pattern. They take as parameters (among others) the user who requested the activity. The application business logic is located in the service layer. Service classes interact with the interface and call the business objects classes. In addition, they call the data access objects that will read or modify the database. Also, the data access layer is in charge of doing the mapping between the database and the application objects. It contains requests and methods that create data transfer objects. These objects are transferred to service classes who create business objects.

Before performing the activity, the services methods perform a security check in order to verify that the activity is allowed by the security policy. If it is prohibited then a security exception is thrown. In order to illustrate this, let's focus on the following example. It shows a method in the service *BorrowerAccountService* that updates the borrower account:

```
public void updateBorrowerAccount(User borrower, BorrowerAccount borrowerAccount, User connectedUser) throws BSException, SecuritPolicyViolationException {

    ServiceUtils.checkSecurity(connectedUser, LibrarySecurityModel.MODIFYACCOUNT_METHOD, LibrarySecurityModel.BORROWERACCOUNT_VIEW, LibrarySecurityModel.DEFAULT_CONTEXT);
```

This method takes as arguments a borrower, its account (modified) and the connected user

(the one who asks for modifying the account). The first instruction makes a call to the method '*checkSecurity*'. This method checks if the security policy allows or prohibits the 'connectedUser' to perform the activity (here '*LibrarySecurityModel.MODIFYACCOUNT_METHOD'*) under the book view ('*LibrarySecurityModel.BORROWERACCOUNT*_VIEW') and in the default context.

The '*checkSecurity'* may raise 2 security exceptions:

- '*SecurityViolationException*': when a security rule is violated. This happens when a prohibited activity is requested.
- '*UndefinedSecurityPolicyException*': when no security rule is defined for the requested parameters.

The code of this function is shown bellow:

```
public static void checkSecurity(User user, Method[] activity, Class view, Context context) throws SecuritPolicyViolationException, UndefinedSecuritPolicyException {
    String result;

    // call the security service
    result = ServiceBO.securitPolicyService.checkSecurityPolicy(user.getRole().getClass(),activity,view,context);
    // it is prohibited
    if(result.equals(SecurityPolicyServiceInterface.PROHIBITION_AUTH))
      throw new SecuritPolicyViolationException("Security policy violation. The requested activity is prohibited");
    // it is undefined
    if(result.equals(SecurityPolicyServiceInterface.UNDEFINED_AUTH))
      throw new UndefinedSecuritPolicyException("undefined security policy behaviour. The response to the requested parameters is undefined.");

  }
```

4.5. Deployment of the security model

In order to deploy the security model, we followed 2 steps:

- Create a generic security policy framework
- Implement the security policy library

The generic framework is not dependant of the implementation. Therefore, it can be used with any OrBAC based security policy. In this section, the security policy UML model is presented. Then, the framework execution and use is shown by a sequence diagram showing how the security mechanism is triggered.

Fig. 9 shows the UML model of the security framework:

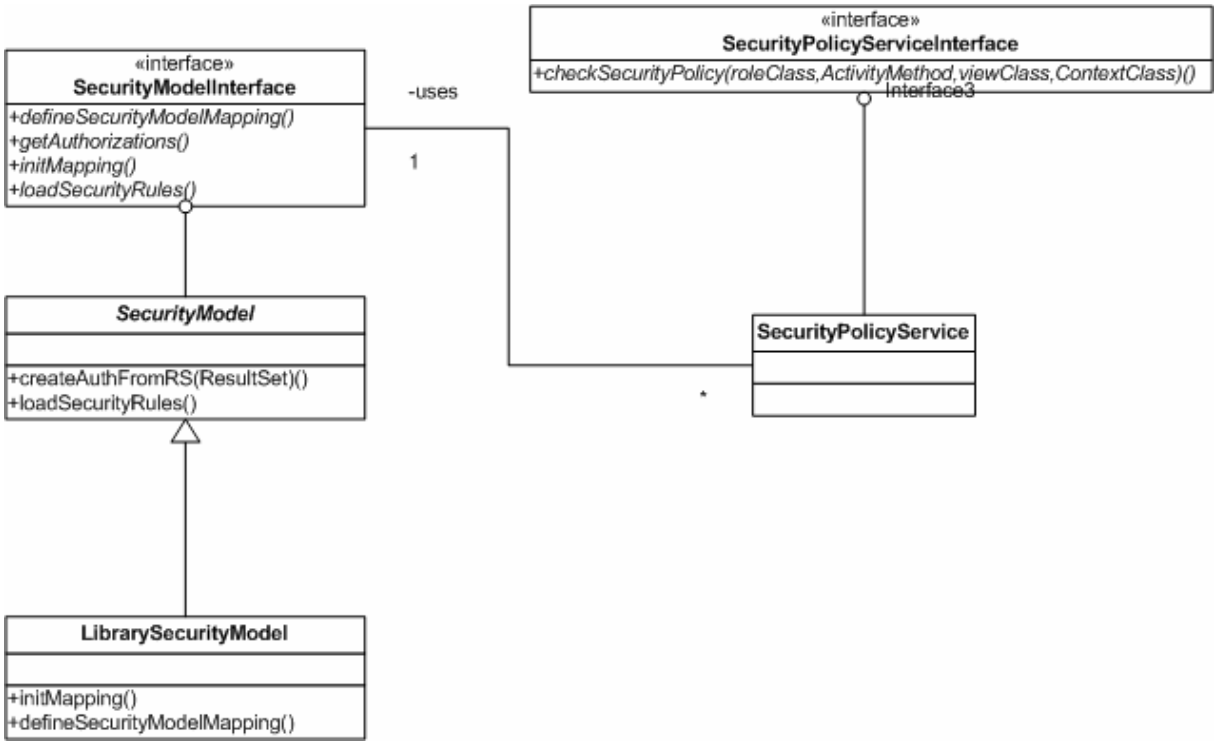

Fig 9. Security framework class model

It is important to point out that the only class that is related to the implementation is "LibrarySecurityModel". This class implements 2 abstract methods:

- initMapping: initializes the constants containing the application classes and methods that will be used to map OrBAC roles, activities, views and contexts.
- defineSecurityModelMapping: defined the mapping between OrBAC entities and the application classes and methods (created by initMapping).

The "SecurityModel" class loads security rules from the database and creates the collection of OrBAC security rules. The only class that will be called by the application is the "SecurityPolicyService" service class. This service checks the security policy in order to find out if the requested activity (knowing the user role, the context and the view) is allowed or prohibited. Fig 10 shows how the security framework is used by the application (when a user tries to borrow a book during holidays which is forbidden by the security policy):

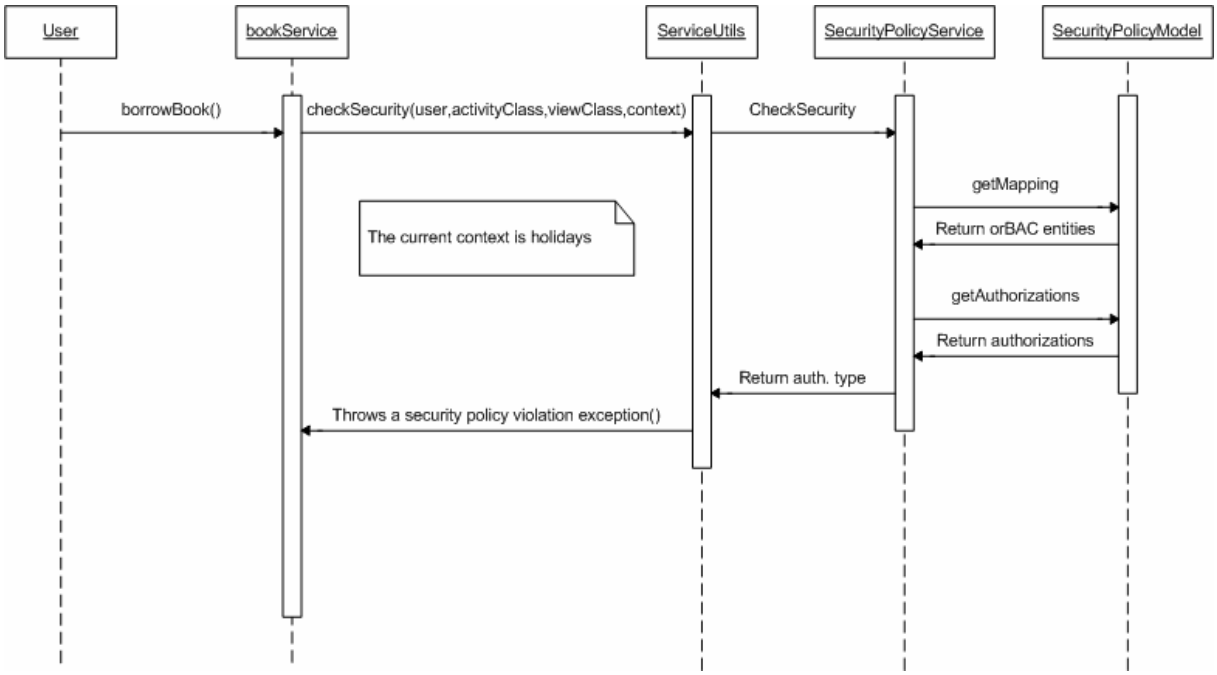

Fig 10. Security policy sequence diagram

All methods of services classes that implement a security policy rule call "checkSecurity" method in "ServiceUtils" before performing the action. Then the security policy service uses the security model to get the OrBAC entities mapped by the parameters. Afterward, the service searches for a rule using the calculated OrBAC entities (in the presented case it is a prohibition rule). Finally, ServiceUtils raises a security policy violation exception when it find out that the requested activity is forbidden.

4.6. Mutation tool

4.6.1. Generation of mutants

The mutant generator is implemented as part of the MotOrBAC tool that implements the OrBAC security model. The goal of this tool is to allow security administrators to specify and define an OrBAC based security policy model.
We added a module that generates security policy mutants. When the security policy is defined the tool creates the security mutants. Its user interface is shown in Fig 12.
.

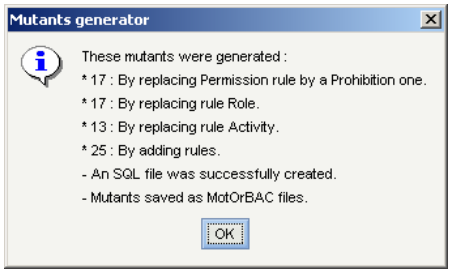

Fig 11. Mutation result dialog box

The user can choose the number of generated mutants for each category. Then the tool creates MotOrBAC security policy files and/or an SQL script that creates a database table containing all the mutant rules (each mutant policy having a unique id).

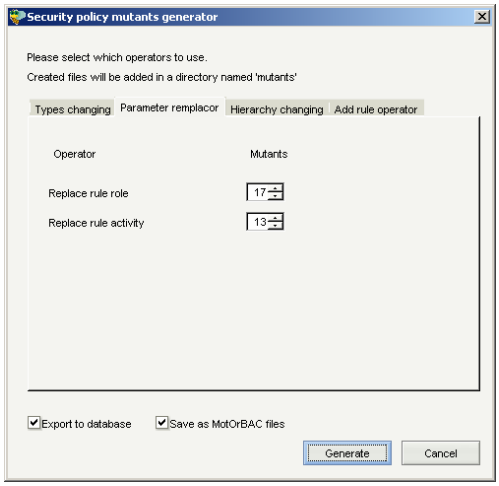

Fig 12. The mutant generator window

The mutation tool uses MotOrBAC libraries to get the current security policy rules list, entities as well as hierarchies. In addition, the mutation tool uses MotOrBAC modules to check if conflicts exist and resolves them. In case of conflict, it means that the mutated rule is in conflict with another. The conflict is automatically solved by giving the highest priority to the mutated rule. The result is that this rule is implemented or executed in priority. The numbers of generated mutants are shown in a dialog box (as presented in Figure 11). For our experiments, we generated all the possible mutants, for ranking the mutation operators.

4.6.2. Oracle function for mutation analysis

In order to decide whether a mutant is killed or not, we use an oracle function that checks the difference between the output of the mutant policy implementation and the correct security policy one. The security mechanism prints the authorization that was granted to the requested action into a file. The oracle function thus consists in comparing the files that are produced by the mutant and the original policy. We present here an example of 2 different outputs:

Output of the application using initial security policy:

```
INFO main root - Permission granted for the requested action reserveBook BORROWER
```

Output of the application using mutant security policy:

```
WARN main root - Requested action prohibited reserveBook BORROWER
```

Information about the operator used to generate this mutant:

```
Operator used: Type Changing Operator

Rule to change:
Permission (rennesLibraries,Borrower, ReserveBook, Book, WorkingDays).

Rule to use instead:
Prohibitionp (rennesLibraries, Borrower, ReserveBook, Book, WorkingDays).
```

This oracle function, by comparing the behaviour of the initial program with the seeded one, is sufficient to determine that a test case kills a mutant. For a practical use, this comparison is not possible and the tester must define an explicit oracle function or manually establish the verdict. This is another reason why it is important to generate only the necessary and sufficient number of mutants.

4.6.3. Generated mutants

Table 3 gives the number of generated mutants per operator. The ANR operator generates much more mutants since it adds a non specified security rule. The number of generated mutants thus reflects the fact that all the possible cases have not been specified in the security policy. To our own experience, this is quite usual: the specification focuses on the most critical and important case, and often considers that a default behaviour is acceptable. Testing these cases allow the default policy to be exercised and allow to highlight lack in the specification. On the other hand, they are few mutants generated from hierarchy changing operators because the specification doesn't introduce many hierarchical entities.

Operator category	Op. name	# mutants
Rule type changing operator	PPR	22
	PRP	19
Rule parameter changing operator	RRD	60
	CRD	60
Hierarchy changing operator	RPD	5
	APD	5
Rule adding operator	ANR	200
All		371

Table 3 - Number of generated mutants per operator

5. OrBAC mutation analysis in action: results and comments

In this section, we present the results of the OrBAC mutation. There are 2 main objectives for this study.

The first objective is to provide a ranking of the mutation operators. This ranking will be based on the mutation analysis results.

The second objective is to study the relation between software test cases and security test cases. In fact, testing functions includes exercising many security mechanisms. The issue is to determine whether testing functions provide enough confidence in the security mechanisms. Then we want to know how to improve this confidence by selecting test cases specific to security. Two strategies are studied for producing security policy test cases, depending if they are built in complement of existing functional test cases or independently from them.

Before presenting the results, we present some definitions in order to help clarifying the subtle differences between functional and security policy testing. After this sub-section, we will start with presenting the experimentation protocol and the strategy used to decide how to rank the operators. Afterwards, in the last sub-section, we study the relation between functional tests and security tests.

5.1. Security tests qualification

Experimental protocol

The *first step* consists of generating minimal test suites per mutation operator, w.r.t. the following definition:

Definition: minimal test suite. A test suite is minimal for a set of mutants iff the test cases it includes have a 100% mutation score and, if when a test case is removed, the mutation score decreases.

We note TS(name of operator) the minimal test suite needed to kill all the mutants generated with this operator. In this study, an important effort has been allocated for generating the test cases and minimizing the test suites. Since it is difficult to have much less than a test case by security rule, we believe the minimal test suites are close from the optimum. For instance, the minimal test suite of 36 test cases selected for killing the basic mutation operators is equal to the number of non generic security policy rules.
The *second step* consists of comparing the mutation operators. This comparison leads to a ranking, which is obtained with two criteria. The first one determines whether a mutation operator can replace another. This aspect is captured by the notion of *subsume relationship*. When two mutation operators are equivalent, any of them can replace the other without loss of efficiency for the generated test cases. To choose which of the two mutation operators can be removed, we consider the number of generated mutants as a second criterion.

Definition: subsume relationship (->). A mutation operator MO1 strictly subsumes MO2 (MO1 -> MO2) if:

a) the minimal test suite TS(MO1) also reaches a 100% mutation score for the MO2 mutants

b) the minimal test suite TS(MO2) does not reach 100% for MO1 mutants.

MO1 and MO2 are equivalent (MO1 <-> MO2) if TS(MO1) reaches 100% on MO2 mutants and TS(MO2) reaches 100% on MO1 mutants.

Fig 13 illustrates the definition. MO1 -> MO2 since the test suite for MO1 is sufficient to kill all mutants generated with MO2. Conversely, MO2 doesn't subsume MO1 since its minimal test suite only kills 80% of the mutants created with MO1. We can thus consider that MO2 can be removed, since it is not needed when qualifying a test suite. MO1 precedes MO2 in the ranking. In case of equivalence, one of the two mutation operators is useless. To determine which one can be removed, we consider that it is better to generate less mutants (due to the execution times, and to the effort needed for generating these mutants when done manually). So, the ranking relation is defined as follows.

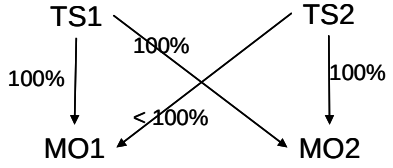

Fig 13. MO1 operator strictly subsumes MO2

Definition: mutation operators ranking (>).

MO1 > MO2 iff: MO1 -> MO2 or **(** (MO1<->MO2) and |{MO1 mutants}| < |{MO2 mutants}| **)**

This ranking only orders partially mutation operators. If a mutation operator is not ranked, it is independent and necessary for a relevant test qualification process.

Ranking of mutation operators

First, the mutation analysis was applied with each minimal test suite for each mutation operator. The results were deceiving since no clear ranking appear. We then consider minimal test suites for couples of mutation operators: PPR-PRP (Rule type changing operators), RRD-CRD (Rule parameter changing operator) and RPD-APD (Hierarchy changing operator). The results for the size of the minimal test suites are displayed in the table bellow. The relationships between minimal test suites are:

TS(RRD-CRD) = TS(PPR-PRP)

TS(RPD-APD) $\subset$ TS(PPR-PRP)

$|$ TS(PPR-PRP) $\cap$ TS(ANR) $|$ = 21 test cases

So, both PPR-PRP and RRD-CRD operators subsume the RPD-APD operator. PPR-PRP and RRD-CRD are equivalent for the subsume relationship. Taking into account the second criterion, the number of generated mutants per operator, we obtain the following ranking:

PPR-PRP -> RRD-CRD -> RPD-APD

This result shows that the PPR-PRP operator should be used in priority, thus avoiding the creation of most mutants.

Operator category	OP	Size of minimal test suites
Rule type changing operator	PPR	36
	PRP	
Rule parameter changing operator	RRD	36
	CRD	
Hierarchy changing operators	RPD	4
	APD	
Rule adding operator	ANR	154

Table 4 Number of minimal test suites

The ANR and PPR-PRP operators are not comparable with this ranking. Some test cases are shared by both minimal test suites (21 test). The table bellow shows the overlap between the test suites, in terms of respective mutation scores. The minimal test suite for ANR kills 59.3 % of the non-ANR mutants. On the other hand, the PPR-PRP test suite only covers 17% of the ANR mutants. The ANR mutants are thus necessary but cannot replace the PPR-PRP operator. PPR-PRP and ANR are not comparable, and are recommended operators. On this case study, the ANR operator is the most costly in terms of generated mutants. This number may vary, depending on the completeness of access control rules. The fewer rules are specified, the more mutants this operator generates. We believe that it is likely that the rules do not specify all the combinations explicitly, and that this operator is the most costly. On the other hand, the PPR-PRP operator will generate more mutants when more rules are added. In a general case, there is thus a balance in the number of generated mutants between PPR-PRP and ANR.

	all mutants except ANR	ANR
TS(PPR-PRP)	100%	17%
TS(ANR)	59.3%	100%

Table 5 Overlap of test suites between PPR-PRP and ANR operators

Removing mutants generating conflicts

To reduce the number of mutants to be generated, we remarked that the mutants which caused a conflict (which is solved by giving priority to the mutant rule) are the more easy to kill. Table 6 presents the number of mutants which generated conflicts per operator. Bellow we present an example of such conflict. The example illustrates that two test cases, generated to kill mutants without conflicts, kill this mutant with conflict.

Operator used: RRD

Initial rule:
Prohibition(Library,Secretary,ManageAccess,PersonnelAccount,Default)

Mutated rule:
Prohibition(Library,Administrator,ManageAccess,PersonnelAccount, Default)

The seeded rule is in conflict with another rule:
Permission(Library,Administrator,ManageAccess,PersonnelAccount,Default)

The two tests that kill this security policy mutant:
Test 1: Test that secretary cannot manage access
Test 2: Test that admin can manage access.

These two tests are already generated to kill mutants without conflicts.

Operator category	With conflicts	Without conflicts
Rule type changing operator	-	41
Rule parameter changing operator	23	97
Hierarchy changing operators	-	10
Rule adding operator	33	167
Total	56	315

Table 6 # of generated mutants with and without conflicts

So, several test cases, from the minimal test suites, systematically kill these mutants. It means that these mutants are not necessary. Indeed, we have:

TS(RRD-CRD with conflicts) $\subset$ TS(RRD-CRD without conflicts)

TS(RRD-CRD without conflicts) corresponds to the suite of 36 test cases needed both for Rule type and Rule parameter changing operators. This test suite is also minimal for the

mutants RRD-CRD which do not cause conflicts. Among this set, only 12 test cases are needed to kill mutants with conflicts which compose the TS(RRD-CRD with conflicts) minimal test suite.
TS(ANR with conflicts) $\not\subset$ TS(ANR without conflicts)
But TS(ANR with conflicts) $\subset$ TS(PPR-PRP) and TS(ANR with conflicts) corresponds to a minimal test suite of 21 test cases (which are the exact intersection of the TS(ANR) and TS(PPR-PRP) test suites.
It means that if the PPR-PRP and ANR operators are used, the minimal test suite of PPR-PRP kills the mutants with conflict generated with ANR. Combining these two operators allows removing the mutants with conflicts generated with the ANR operator. Around 18% of the mutants (those with conflicts) can be removed without any loss of relevance for the generated test cases.
We study why these mutants could be removed and the explanation is interesting, since it corresponds, for security testing, to the 'coupling effect' analyzed by Offutt et al. [14]. In fact, when a mutated rule causes a conflict, it has an impact on at least two rules: this fault is equivalent to two sequential mutations without conflicts.
In conclusion, some operators are more relevant than others for improving the quality of security test cases. We present a ranking of the most useful mutation operators:

1. Adding rules operators
1. Rule type changing operators
2. Rule parameter changing operators
3. Hierarchy changing operators

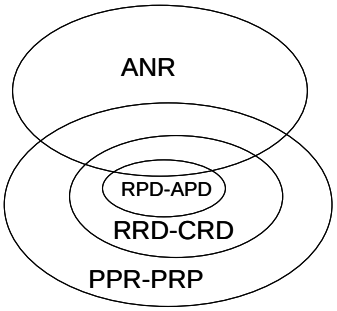

Fig 14. Relation between operators

Fig 14 displays this ranking, and highlights the overlap between ANR and the other operators. All these operators generate mutants which intersect with the adding rule operator. Adding rules operators are the most interesting because they simulate cases that are not tested by functional tests. As shown by the results, they are the most difficult to kill, regarding the number of test cases needed to detect its generated mutants. Only advanced security test cases are able to kill mutants without conflicts created by this operator.

5.2. Security tests vs. functional tests

Two test criteria are studied in order to select test cases from an OrBAC security policy model.

CR1 - *The criterion 1 (CR1) is satisfied iff a test case is generated for each primary access control rule of the security model.*

In the case of the LMS, we have 20 such primary rules. In the case of a generic rule, a test case testing one instance of this rule is considered as sufficient.

CR2 - *The criterion 2 (CR2) is satisfied iff a test case is generated for each primary and secondary rule of the security model, except the generic rules.*

In that case, 35 test cases are generated corresponding to the 42 total access control rules minus the 7 generic ones. The CR2 criterion is stronger than CR1, since it forces the coverage of all secondary rules.

Advanced SP test cases - *A*dvanced SP test cases *that exercise the default/non specified aspects of the security policy.*

These test cases are selected to kill mutants generated with a specific mutation operator (ANR) presented in the next section.

Functional test cases: *It corresponds to system tests, in the sense they are generated based on the use cases of the system under test.*

We used the approach based on use cases + contracts (pre- and post-conditions) to generate the functional test cases. The automated generation is obtained using the UCTS (Use Case Transition System) presented in [10]. The generated test cases cover the nominal code (code implementing the specified use of use cases) and a part of the robustness code of the system under test (unexpected use of a use case and specified situations when the use case execution fails).
Security test cases obtained with CR1 or CR2 should test aspects which are not the explicit objective of functional tests, e.g. that all prohibition rules that are not tested by functional tests.

Test strategies

We study whether the functional test cases can be used for security policy testing. Reusing functional test cases implies adapting them for explicitly testing the security policy. The intent of the functional test becomes *security* and details the SP rules which are tested by the input test sequence. The test oracle does not check the correctness of the service results, but interrogates the security mechanism and checks if the expected permission/prohibition is executed.
So, we consider two types of strategies depending whether we reuse the existing test cases or not.

Incremental strategy: *It denotes the strategy for producing security test cases which reuse existing test cases.*

An example of incremental test strategy consists of reusing functional test cases, then completing them with one of the CR1 or CR2 criteria, and finally completing the resulting test suite with advanced test cases.

Independent strategy: *This strategy consists of selecting functional, SP test cases and Advanced SP test cases independently.*

We will compare and discuss in the case study several incremental strategies and the independent one. The goal is to highlight the many issues that arise when selecting test cases for the security policy aspect.

Issue 1: Relationships/differences between functional tests and security ones

The first experiments we performed aim at studying whether the faults detected by functional test cases differ from (are included in or intersect with) the ones detected by test cases dedicated to the security policy. While the functional tests, which cover 100% of the code, necessarily execute security mechanisms, they should only focus on the success/failure of method/service sequence executions.

When reusing functional test cases with the objective of testing security, the intent changes and thus it is necessary to modify the associated test oracle. This task may be costly in the general case, depending on the difficulty to relate the functional test sequence to the security mechanisms it should exercise.

Table 8 summarizes the number of functional test cases. The table shows that some test cases do not trigger the security mechanisms. It corresponds to the execution of code which is not related to the security policy. It confirms the intuition that functional test objectives differ from the objective of explicitly testing security mechanisms.

Table 7 second column shows that the functional test cases, adapted to security, can kill most (78%), but not all, basic operators' mutants. Concerning, the ANR operator, the functional test cases are not efficient (11%): it is due to the fact that the ANR operator generates security flaws which are outside the scope of the specification. In conclusion, the overlap of functional aspect and security policy is high, but the functional test cases do not kill all security mutants. It appears as a meaningful task to generate test cases with the explicit objective of testing the security policy.

	Functional		**CR1**		**CR2**	
#Test Cases	42		20		35	
Rule type changing operator	35/41	85%	38/41	92%	41/41	100%
Rule parameter changing operator	90/120	75%	101/120	84%	120/120	100%
Hierarchy changing operators	10/10	100%	10/10	100%	10/10	100%
Overall score with basic security operators	135/171	**78%**	149/171	**87%**	171/171	**100%**
Rule adding operator (ANR)	22/200	**11%**	28/200	**14%**	33/200	**17%**
Overall score with all operators	157/371	**42%**	177/371	**47%**	204/371	**55%**

Table 7 Mutation analysis results by test cases category

	# test cases
Tests that do not that trigger sec. mechanisms	7
Tests that trigger security mechanisms	35
All tests	42

Table 8 Functional tests analysis

Issue 2: Comparing test criteria

Table 7 third and fourth columns present the mutation results for CR1 and CR2 test criteria. The 20 test cases selected with CR1 are more efficient than the functional test cases, since

with fewer test cases the final mutation score is higher. However, this criterion is not efficient to reach a 100% mutation score on basic mutants (generated with all operators except ANR).
In conclusion, the CR2 criterion is necessary to provide a full coverage of basic mutants, and appears as good trade-off between functional and CR1, in terms of efficiency (mutation score) and cost (#test cases) for detecting security flaws. A corollary conclusion is that a large number of basic mutants are quite easy to kill. Since the hard-to-kill mutants are the most interesting ones (because they require the most efficient test cases), another study would consist of ranking the mutation operators so that only hard-to-kill mutants are generated. This study is beyond the objective of this report but is necessary to offer a realistic mutation-based approach to test security policies. This report focuses on the test selection problem for security policy and the question of sufficient mutant operator does not impact the conclusions we draw (in the worse case, we consider more mutants than necessary since some mutants are coupled).
We remark that the CR2 criterion allows covering up to 17% of the ANR mutants. While basic security mutants force the tests to cover the specified security rules, the ANR ones are useful to check the robustness of the system in case of default or underspecified policies. Combining both operators provides a good criterion to guide the tester when generating the test cases.
The *advanced security test cases* are test cases which are explicitly generated in order to kill all the ANR mutants. In this approach, we are using mutant score as the test criterion. We do not use mutation for analysis purpose (to compare test criteria or testing technique) but as a test selection technique. In the case mutants generation cannot be fully automated, it makes this test selection technique impossible to apply. However, the study we propose provides interesting results, in terms of test selection effort and test quality.

Issue 3: Advanced vs. basic security tests
The issue here is to compare the test cases selected to cover the security rules with CR2 (and which kill all mutants except ANR) and the advanced security tests which are generated in order to kill all the ANR mutants. Table 9 presents the overlap between these two approaches. It is interesting to note that the advanced security test cases kill up to 60% of the basic security mutants. On the other hand, the test cases selected with CR2 only kill 17%. The effort to kill the ANR mutants is much more important (154) than for killing the basic security mutants (35).

	#test cases	**Basic security mutants**	**ANR**
CR2	35	100%	17%
Advanced sec. tests	154	59.3%	100%

Table 9 Overlap of CR2 and advanced test cases

In conclusion, the test selection based on the ANR mutants cannot replace the CR2 criterion. CR2 and advanced security test cases are not comparable, and are both recommended, the first to efficiently test the specification, the second to cover non-specified cases (robustness).
Issue 4: Incremental vs. independent test strategies
The issue now is to study whether we can leverage an incremental approach to save test generation effort. Table 10 recalls the number of test cases generated with the following strategies: the independent approach (we do not reuse functional test cases) reusing the functional test cases, completing them to reach the CR2 criterion and to kill all ANR mutants

(*Incremental from functional strategy*), generating test cases to reach the CR2 criterion, completing them to kill all ANR mutants (*Incr. from CR2 strategy*).
The first incremental strategy seeks to take benefit from the existing functional test cases (which have to be adapted for security), the second one starts from the CR2 test criterion.
Even if quantitative results are displayed, the comparison is difficult because the effort to adapt functional test cases to security cannot be easily estimated in a general case. It depends on the system to be tested. It may be neglected: that's the case for our study since the security mechanisms are centralized and can be quite easily observed. In a general case, adapting these test cases may be as costly as generating security test cases. In Table 11, we put two values that correspond to the case when the cost of adaptation can be neglected in parenthesis. This issue is related to the problem of security mechanisms testability (controllability and observability).

#Test cases / Strategy	**Funct.**	**Basic Security CR2**	**Advanced Security**	**Total**
Independent	-	35	154	189
Incremental from functional	(42)	21	133	196 (154)
Incr. from CR2	-	35	133	168

Table 10 Independent vs. Incremental test generation strategies

It appears that the independent generation of basic security and advanced test cases is the most costly. With any incremental strategy, we need to generate 133 test cases to kill all advanced security mutants (saving 21 test cases generation). The final ranking between the two incremental strategies depends on the adaptation cost of functional test cases and may vary from a system under test to another. Only the test experts may estimate this adaptation cost. If it can be neglected, reusing functional test cases and completing them is the most interesting strategy. Another decision has to be taken which is to put an important effort for testing the security policy robustness (killing ANR mutants).

5.3. Conclusion for the use of mutation analysis for SP testing

The objectives of the study are two-fold:
First, it proposes mutation operators for security policy testing. To qualify a set of security policy test cases, a classical mutation analysis is applied with these 'security' mutants. In practice, it may be costly to implement all types of security mutants, especially if it must be done manually. We thus performed detailed experiments, as rigorously as possible, on a 3-tiers architecture example (a library system) to select a sufficient subset. The hardest-to-kill security mutant operators are those which must be generated in priority. If this initial study has to be completed with others, the first results reveal that we can limit the types and number of generated mutants. The ranking shows that two operator types are necessary: rule type changing operator (PPR and PRP) and rule adding operator (ANR). Combining these two operators also leads to another minimization of the generated mutants: only mutants which do not cause conflicts in the security rule have to be created. It is due to the fact they introduce at least two coupled elementary faults (coupling effect). While PPR-PRP mutants force the test to cover the specified security rules, the ANR ones are useful to check the robustness of the system in case of default or underspecified policies. Combining both operators provide a good criterion to guide the tester when generating the test cases.

Secondly, the study focused on comparing functional and security policy testing, and on estimating the interest of two test selection criteria and two test strategies. The study does not answer the many issues security policy testing tackles but allows underlining them and shows that testing a security policy is a specific task, which is complementary from the functional testing one. We proposed a first methodology to select test cases, with various test selection criteria based on the security policy (specification) or on mutation (code based). Finally, we distinguish test selection for testing the "nominal" security policy rules from the advanced test selection that aims at testing the robustness of the security policy. The first test cases can be derived from CR2 test criterion. We use ANR mutants as a suitable criterion for testing security policy robustness. To conclude concerning the use of mutation as a sSP testing technique, the studies must be balanced by considering wo qualitative aspects that makes the approach feasible or not:

- the possibility, or not, to adapt functional test cases to test a security policy,
- the interest of advanced security tests regarding the important additional effort it may require.

6. Mutation analysis applied to SQL Injection

A lot of web applications use back end databases. SQL injection is one of the simplest and well known attacks that can easily be used to damage or to perform illegal actions against these web applications. However, it is surprising that many web applications are vulnerable to this attack. It is very easy for hackers to find vulnerable websites and attack them using the SQL injection technique.

In this section we study how to adapt mutation analysis to the context of SQL injection attacks. We start with a brief presentation of SQL injection attack. Then we present the most interesting approaches used to cope with SQL injection attacks. Afterwards, we propose a strategy to adapt Mutation analysis to qualify SQL injection countermeasures and to evaluate the testing tools. Afterwards, the results of mutation analysis applied to SQL injection countermeasures are presented. Then, based on the results and the mutants, we propose some improvements that can be applied to SQL injection countermeasures to overcome the limitations exposed by the mutation analysis study.

6.1. SQL injection attacks

SQL injection attacks exploit the fact that some web applications do not perform the input data validation, and create SQL query strings dynamically. The aim of SQL injection attack is to append SQL statements inside the application SQL query and execute a different query.
In order to understand the SQL injection, let us focus on the following example, this code is used inside a java class. (It is a "hello world" example always presented by reports to show the SQL injection technique):

*String unsafeQuery = "Select * FROM users WHERE user = '" + inputUser + "' and password = '" + inputPassword + "';";*

A rogue user (or a hacker) may type this as *inputUser* and *inputPassword*:

InputUser? ' or 1=1; --'

inputPassword? Nothing

Now we end up with the following query:

*unsafeQuery = "Select * FROM users WHERE user = '' or 1=1; --' and password = 'Nothing';*

The last part of the query that starts from "--" is ignored. It is considered as comment because the string "--" indicates a comment. The query that will be executed is this one:

*Select * FROM users WHERE user = '' or 1=1;*

This query is a tautology and will return all users. In this case, the attacker succeeded to log in (illegally).

6.2. Countermeasures security mechanisms against SQL injection

6.2.1. Basic security mechanisms based on input validation

Many approaches were proposed to deal with SQL injections based attacks. Input validation is an obvious approach that is easy to implement. There are 3 different options for input validations [15]:

- Massage data to get valid data.
- Reject illegal characters.
- Accept only authorized characters

data massage consists on escaping data. For example if the user types: [' *or 1=1*--]. This string becomes: [*'' or 1=1*] or [*\' or 1=1]*. The query will be executed as expected and there will be no harm. This technique has drawbacks. The list of suspicious data is not finite and new suspicious characters are added all the time (it depends on new versions of SQL database that may use new reserved words or on hackers who invent new techniques etc…). For example, SQL server databases uses the character '#' to delimit dates. This character was not in the list of suspicious characters and hackers used it to perform attacks.

Furthermore, the developers may limit the input data length. In this case, data massage is insufficient. Here is an example [15]:

If the user name length is limited to 16, a user may type (see that inputUser contains 16 characters, it is done intentionnaly):

inputUser? aaaaaaaaaaaaaaa'

inputPassword ? '; *shutdown;* --

The application will try to escape the quote located at the end of inputUser. The problem is that the escape character is ignored since the user name length is limited to 16. The query becomes.

*unsafeQuery = "Select * FROM users*

WHERE user = 'aaaaaaaaaaaaaaa'' and password = ''; '; shutdown; --'

This query shutdowns the server. Because it looks for user name = [*'aaaaaaaaaaaaaaa'' and password = ''; '*]. Then the *shutdown* command is launched (a dangerous attack reached its goal).

The technique of rejecting illegal data consists of removing all suspicious characters used as input. The string will not contain characters like "'" or "--" that may be used to perform an SQL injection attack. The main drawback of this approach is that we do not know all possible dangerous characters (as explained above).

The last technique (accepting only authorized input) consists of accepting only letters and numbers. This is the most efficient technique because it makes impossible an SQL injection attacks since the input data will never contains suspicious characters.

The main drawback of input validation is that it is prone to a many false negatives. For example, a user can type his name (say) O'Brian. In this case, the application may consider the input data as an attack since it contains the bad character (‘). In addition, as explained above this technique does not guarantee that there will be no false positive because there is no complete and final list of bad data. Furthermore, we know that if the third technique (Accept only authorized characters) is implemented the application will be completely immune to SQL injection attacks. This is obvious because only letters and numbers are accepted. The disadvantage of this approach is that application users (whose names are for example O'Brian, O'Connor or Bassington-Bassington) may refuse to have their name written differently (and become OBrian or O Brian). Moreover, strongly restricting input data sometimes makes no sense. Why would we forbid that users search for things like "director_at_company.com" (special character used), "2001/12/03" (no alphabetic characters at all) and so on?

6.2.2. Advanced security mechanisms

Many approaches were proposed in the literature. We present here some of the most interesting techniques used to protect from SQL injection attacks (SQLIA).

SQLIA detection: combining static and runtime analysis

In [16], the authors propose a technique to detect SQLIA. If an SQLIA is detected the application rejects the query and does not execute it. A static analysis tool is used to create an *SQL Finite State Machine* (SQL-FSM) for each query. Figure 15 shows an example of SQL-FSM [16] (the query represented is "*Select * FROM user WHERE login = ‘" + input "’*"). SQL-FSM for applications programs are constructed off-line, not at runtime.

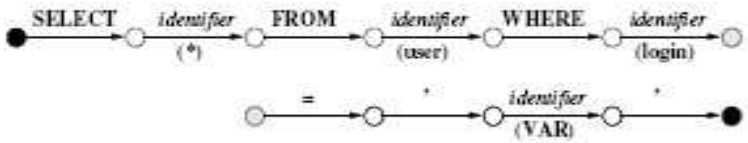

Fig 15. SQL-FSM

During runtime, SQL-FSM representing the executed query (containing input data, a potential SQLIA) is created then compared with the SQL-FSM for that query that was created before (off-line). If the dynamically created SQL query does not conform to the expected query then it is rejected (An attempt of SQLIA). Figure 16 shows an example SQL-FSM violation (from [16]). The input data is [‘ *or 1=1* --] .

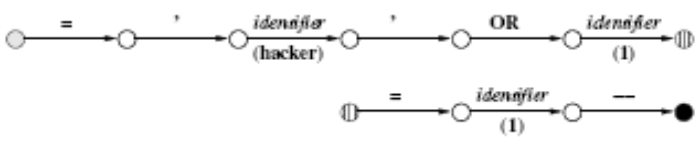

Fig 16. SQL-FSM violation

The main drawback of this approach is the additional runtime analysis overhead. The authors propose solutions to overcome this problem by reducing the number or scanned queries.

SQLRand: a radical shift

This technique was proposed by Stephen W. Boyd et al. [17]. They apply instruction-set randomization against code-injection attacks (mainly buffer overflow based attacks) to the problem of SQL injection. The idea behind the concept is to create randomized instances of the SQL query language by using random keywords. The SQL keywords (like select or update etc…) are appended by a random number called a key. All application queries are randomized inside code scripts or sources. Then a *de-randomizing proxy* is used to recover the query (the proper SQL syntax) and send it the database management system. The malicious input is rejected by the proxy parser, as it uses unknown keywords (the assumption is made that the user will not be able to guess the value of the key).

For example, we have the following query:

SELECT gender, avg(age) FROM cs101.students WHERE dept = '$dept' GROUP BY gender

To randomize the query, a utility appends automatically the key (here it is 571):

SELECT571 gender, avg(age) FROM571 cs101.students WHERE571 dept = '$dept' GROUP571 BY571 gender

A rogue user may attempt to type "' *or 1=1;* --" the query became :

SELECT571 gender, avg(age) FROM571 cs101.students WHERE571 dept = '' or 1=1 ; -- GROUP571 BY571 gender

The SQLIA fails because when the parser tries to parse this query the "or" is identified as unknown keyword since it does not contain the key "571".

The figure 17 presents the architecture proposed by SQLRand approach (from [17]) :

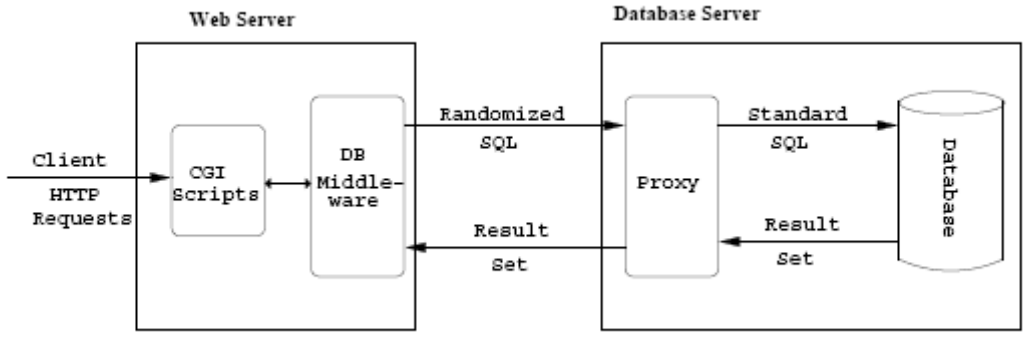

Fig 17. SQLRand architecture

The drawback of this approach is that it requires developers to adopt a new programming

paradigm. Moreover, it is difficult to apply to legacy software as it requires the medication of all applications queries.

String tainting based techniques

Several approaches relying on tainting are proposed in the literature. We present one of these approaches named positive tainting that was proposed by Halfond and Orso [18].

Positive tainting consists on marking query strings hard-coded in the applications tainted as safe. Application strings are identified as trusted. Then the user input strings added by the user are tainted unsafe. A new library called MetaString is used to extend the behaviour of Java Standard classes. The MetaString objects mark the safe characters in the string.

This allows to check the part of query added by the user to test that no SQL reserved words are injected into the query.

This allows to detect malicious SQL code injected to the query.

This approach works as follow:

- All trusted strings are marked (hard-coded parts only are marked trusted)
- Before executing a query a test is done. It checks that all reserved words are tainted as trusted.

In order to illustrate this approach the Figure 18 presents a high level overview of this system [18].

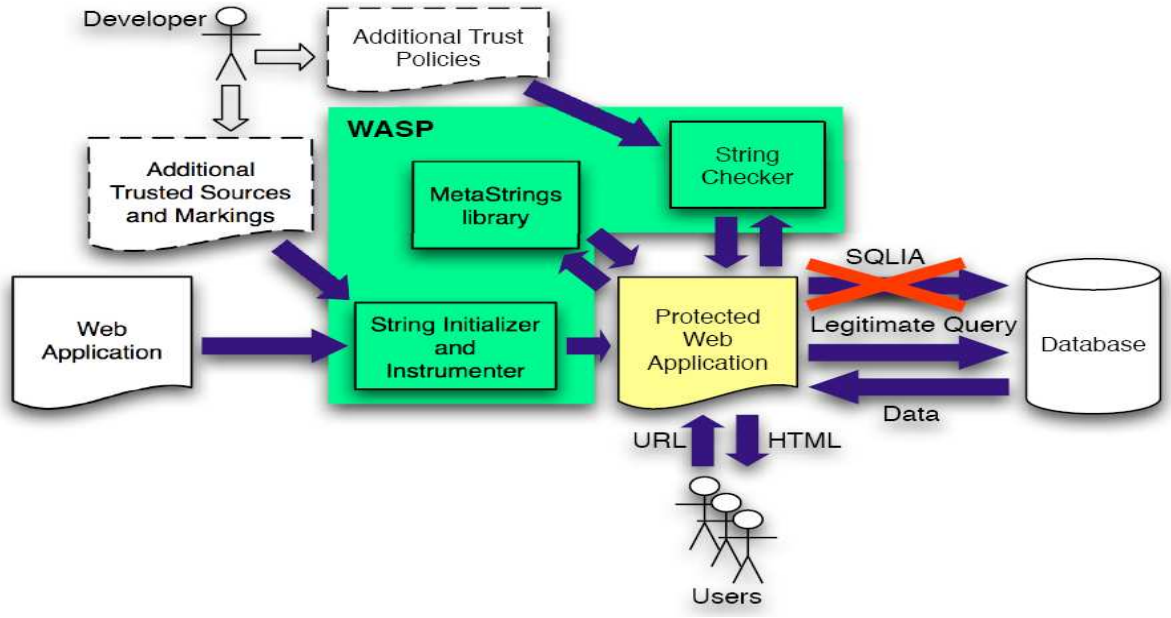

Fig 18. Overview of positive tainting approach

This technique was implemented in a tool named WASP targeting only applications written in Java . The tool uses a library to modify the Java bytecode files by replacing all String declarations by a new class named MetaString that marks as tainted the hard-coded strings and leaves untainted the dynamically added data. In addition, the bytecode classes are scanned to find all database calls in order to invoke a Metachecker class that tests the query before authorising the execution of that query. The Metachcker uses an SQL parser that the query is valid.

A learning based approach

This approach [19] is based on the idea that an application always performs the same types and models of SQL queries and if a new kind of query is performed then this is an indicator of a possible SQLIA.
This approach works as follows:

- During a training period the security mechanism learns all the models of queries that can be performed by the application.
- The security mechanism monitors checks every query performed by the application to see if it is linked with an already known model. If it is not then an alert is raised.

The security mechanism is located between the application and the database server. Therefore, if is completely independent from the application language and platform. The figure 19 presents an overview of the system (from [19]).

One limitation mentioned by the authors for this approach is that it might not detect attacks when the structure of the malicious query matches the structure of another accepted query. In this case, the attacker changed the query in a way that makes it similar to another query normally used by the application. The security mechanism is not able to detect this attack. This potential shortcoming can be overcome by making the association at a finer level.

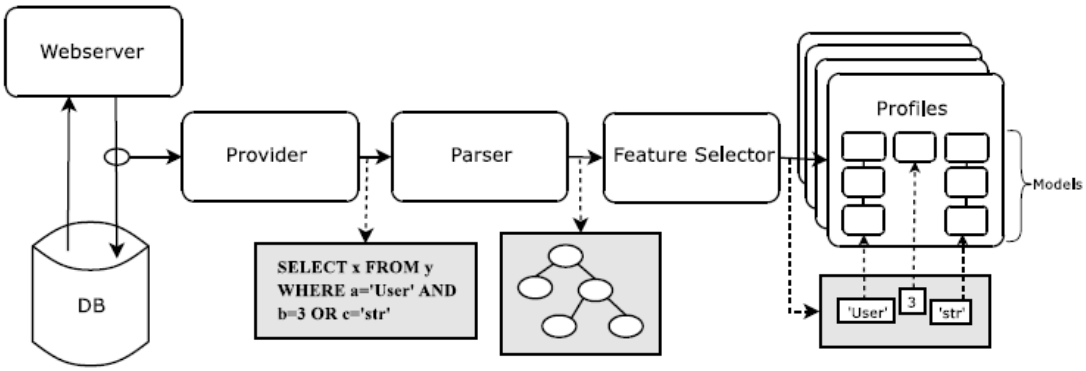

Fig 19. Overview of the learning based approach

6.3. Qualitative criteria for security mechanisms comparison

We introduce a comparison of the approaches we presented above. We use the following criteria to compare the different techniques:

- Deployment cost:
  - High cost: manual modification of application code.
  - Medium cost: semi automatic deployment.
  - Low cost: fully automated deployment.
- Runtime overhead
  - High: important overhead.
  - Low: neglected overhead.

- Complexity of the approach
  - Complex: the use of the technique requires a lot of effort and knowledge.
  - Not complex: the technique is simple to use and deploy.
- Portability
  - Platform and language independent.
  - Language dependent.

In the following table, we present this comparison:

	Deployment cost	Runtime overhead	complexity	Portability
Input validation	High	Low	Not complex	No
Static and runtime validation	Low	Low	Complex	No
SQLRand	Medium	Low	Complex	Yes
Tainted string	Low	Low	Complex	No
learning machine	Low	Low	Not complex	Yes

Table 11 Comparison of SQLIA countermeasures

6.4. Fault injection applied to SQLIA

6.4.1. Adaptation for SQLIA countermeasures

In order to test the proposed techniques against SQLIA, we need a security fault model that is adapted to the context of SQLIA.
We propose to apply a technique similar to what was proposed by Mathur et al. concerning environment perturbation [5]. The purpose of this approach is to evaluate the robustness of the mechanism when its environment is behaving badly.
Environment perturbation can be done in the following places:

Database: The database scheme and the database behaviour. For example:
- making the database show all error messages.
- Introducing unsafe behaviour in the stored procedure by using execSQL that open the door to SQLIA.

The application code:
- Weakening the application code by making it rely only on the security mechanism.
- Making the application print the error message coming from the database.
- Modifying the application behaviour in a way that is different from the security mechanism assumptions. For example, some security mechanisms make the assumption that the query is constructed by making concatenation of string. So it is interesting to see if the security mechanism works when this is done differently.

6.4.2. Existing testing techniques

WAVES [20] is a framework for testing web application in a transparent way. 2 types of security problems are addressed, SQLIA and Cross site scripting. The testing technique consists on trying to perform an attack on web pages and then observing the behaviour of the web application. An enhanced crawler is used in order to retrieve the forms input and then injects malicious data.

SecuBat [21] is a tool that allows to test web applications to check if they are vulnerable to SQLIA. Basically, SecuBat is a web vulnerability scanner. The authors tried it against popular websites. The tool was able to find potential vulnerable websites, among them companies and a finance ministry website.

6.4.3. Adaptation for testing tools

In order to evaluate SQLIA testing tools, we can use the following strategy:

- Checking in the tool works on unprotected application.
- Removing protection randomly from only one form input in order to see if the tool uses all users input.
- Removing protection randomly from only one web page in order to see if the tool uses all application web pages.
- Introducing SQLIA vulnerability in unusual places, for example, in the program that handles the cookies and generates queries from cookies content. This will allow to see if the tool is capable of covering different types of SQLIA sources.

6.4.4. Combining both adaptations

It is interesting to combine both adaptations. this means weakening one security protection and evaluating the testing tool to see if it succeeded to benefit from the security breach created in the protection mechanism

6.5. Results of mutation analysis study applied to SQLIA

6.5.1. Input validation

We apply traditional fault injection technique for input validation since its context is very similar to software testing. The fault injection strategy consists on focusing only on the input validation code. Then traditional mutation operators are applied. The final goal of this technique is to improve the security tests used to evaluate the security mechanism.

6.5.2. Adaptation of fault injection for SQLIA advanced countermeasures

In order to test the proposed techniques against SQLIA, we need a security fault model that is adapted to the context of SQLIA.

We propose to apply a technique similar to what was proposed by Mathur et al. concerning environment perturbation [5]. The purpose of this approach is to evaluate the robustness of the mechanism when its environment is behaving badly.

Environment perturbation can be done in the following places:

Database: The scheme of the database and its behaviour. For example:

- making the database show all error messages.
- Introducing unsafe behaviour in the stored procedure by using Exec(SQLQuery) that opens the door to SQLIA.

The application code:

- Weakening the application code by making it rely only on the security mechanism.
- Making the application print the error messages coming from the database.
- Modifying the application behaviour in a way that is different from the security mechanism assumptions. For example, some security mechanisms make the assumption that the query is constructed by making concatenation of string. So it is interesting to see if the security mechanism works when this is done differently.

6.5.3. Adaptation for testing tools

In order to evaluate SQLIA testing tools, we can use the following strategy:

- Checking in the tool works on unprotected application.
- Removing protection randomly from only one form input in order to see if the tool uses all users input.
- Removing protection randomly from only one web page in order to see if the tool uses all application web pages.
- Introducing SQLIA vulnerability in unusual places, for example, in the program that handles the cookies and generates queries from cookies content. This will allow to see if the tool is capable of covering different types of SQLIA sources.

6.5.4. Combining both adaptations

It is interesting to combine both adaptations. this means weakening one security protection and evaluating the testing tool to see if it succeeded to benefit from the security breach created in the protection mechanism.

6.5.5. Applying fault injection to SQLIA countermeasures

Before checking each type of SQLIA countermeasure, let us choose some mutants. We will define two types of mutants :

- General mutants: They will be used for all approaches.
- Specific mutants: They are adapted to the approach. They are made in order to behave in a way that will perturb the security mechanism.

Specific mutants will be presented in the each approach section. The following general mutant are defined:

- Mutant 1: The application displays all errors messages coming from the database. This is the default behaviour of the application. This case is realistic because sometimes the

developers forget to switch off the option that displays errors and debugging information in web pages.

- Mutant 2: The application relies only on the security mechanism. No sanitizing function is used to filter user input. malicious character are accepted which opens the doors to SQLIA.
- Mutant 3: The application uses a stored procedure that contains unsafe behaviour in it. It uses exec(SQLQuery) (that is defined in SQL Server databases and is vulnerable to SQLIA).

SQLRand

We test this approach by creating the following specific mutants:

- Mutant S1: The application constructs some queries by reading files that contain those queries.
- Mutant S2: The application uses stored procedures and dynamically created queries. Some application contain many modules that were developed by different teams so it is possible that some choose to use stored procedures and other choose to created directly the SQL queries in the code.

In the following table, we present the results of the fault injection applied to SQLRand

	Mutant 1	***Mutant 2***	***Mutant 3***	***Mutant S1***	***Mutant S2***
Mutant killed?	No	Yes	No	No	No

Table 12 Fault injection applied to SQLRand

Combining static and runtime analysis approach

For this approach we define the following mutants:

- Mutant S1: The application creates dynamic queries. The user is allowed to create his own queries or sub queries.
- Mutant S2: The application uses a function to call the database pre-defined API (like Statement.execute(query)) and do not call directly this function after constructing every query.

In the following table, we present the results of the fault injection applied to this approach:

	Mutant 1	***Mutant 2***	***Mutant 3***	***Mutant S1***	***Mutant S2***
Mutant killed?	Yes	Yes	No	No	No

Table 13 Fault injection applied to static and runtime analysis

Learning based approach

The following mutants are defined for this approach:

- Mutant S1: The application modules generates a lot of queries. This assumption is realistic because a large application may perform hundreds of queries.

In the following table, we present the results of the fault injection applied to this approach:

	Mutant 1	***Mutant 2***	***Mutant 3***	***Mutant S1***
Mutant killed?	Yes	Yes	Yes	No

Table 14 Fault injection applied to learning based approach

String tainting based techniques

For this approach we define the following mutant:

- Mutant S1: The application constructs some queries by reading files that contain those queries.

In the following table, we present the results of the fault injection applied to this approach:

	Mutant 1	***Mutant 2***	***Mutant 3***	***Mutant S1***
Mutant killed?	Yes	Yes	No	No

Table 15 Fault injection applied to string tainting approach

6.5.6. Summary of mutation analysis results

Technique/ Results	***Mutant 1***	***Mutant 2***	***Mutant 3***	***Specific mutants***
SQLRand	No	Yes	No	0/2
Static and runtime analysis	Yes	Yes	No	0/2
Machine learning	Yes	Yes	Yes	0/1
String tainting	Yes	Yes	No	0/1

Table 16 Summary of fault injection applied to SQLIA countermeasures

Table 16 present a summary of the mutation analysis results. It appears according to the study that the best approach is the learning based approach because it kills 3 mutants out of 4. In fact, it is the ideal approach because there is no way to bypass it since it knows the queries that the application perform and consequently it is not possible to perform unknown queries.
In the next section, we propose some improvements based on the mutation analysis results. The purpose of these improvements is to increase the mutation score of these techniques and by the way overcome their drawbacks.

6.6. Improvements of SQLIA countermeasures

In this section, we present some improvements of the SQLIA countermeasures in order to overcome the limitations highlighted by the mutation analysis study. Combining techniques is a good strategy in order improve the results. After presenting the improvements, we present how to combine approaches.

6.6.1. Improvement of SQLRand technique

We suggest the following improvement. We can perform randomization to user input code. This means dynamically randomizing the input user string when it contains SQL keywords and commands.

Example:

SQL query: *"Select * FROM users WHERE user = '" + inputUser + "' and password = '" + inputPassword + "';";*

User name entered: *inputUser* = "*' or 1=1; --*"

i*nputPassword* = "blablabla"

The user input become after randomization (when the key is 571) : "*' or571 1=1; --*"

The query becomes then:

*Select * FROM users WHERE user =' ' or571 1=1; -- and password =* 'blablabla';

When the query is executed an error occurs. It is important to note that when the user types legal input no randomization is done and the query is executed normally.

6.6.2. Improvement of learning based approach

We can make association in a finer level. The finest level is where the query is executed. the hotspots points where the execute function is called.
The steps to follow :

- Locate all hotspots.
- Before executing the query perform a call to a checker. The checker contains a list of models corresponding to all queries normally executed from that location.

6.6.3. Combining advanced approaches

In order to benefit from the advantages of each technique and limit the drawbacks, we can think about combining 2 approaches. We present here some interesting ideas that can be investigated deeply.

A Learning system and SQL randomization
The idea here is the combine the learning technique with SQLRand. The SQLRand will be applied as presented in the improvement proposal. This means that randomization is applied only the user injected parts of the query. In order to locate, the dynamic part of the query, we use a learning based technique. The system learns what are the parts or the query that change, Afterwards, it stores that information and then in the detection phase will randomize the dynamic parts of the query according to the known during the learning phase.

Tainted string SQL and SQL randomization
Tainted string can be used instead of the learning system. part of the query that contain the user input are tainted, then they are randomized before being executed. SQLIA will cause the query to crash due to randomization.

6.7. Towards SQLIA robust security mechanisms

This study of security mechanisms robustness related to SQLIA is still very incomplete but provides a first step in order to improve these mechanisms. The philosophy we applied to conduct this comparison is similar to the one we applied for security policies. However, in the case of SQLIA, the mutants we proposed are strongly connected to the security mechanisms and are used as a method to detect weaknesses. The improvement of the security mechanisms has been proposed when a potential weakness has been detected. This work can be completed by studying other contexts where mutation analysis and other testing techniques such us the bacteriological algorithm can be applied.

7. Conclusion

In this report, we presented a new approach to deal with security testing. Our approach is inspired by mutation analysis which is a testing technique that proved its effectiveness in software testing. We adapted this technique to two security contexts.

Firstly, we applied mutation analysis to the access control security model (OrBAC). the fault model (mutation operators) are defined at high level, to the ORBAC specification) and then is mapped to the implementation of the security model. We also identified a list of mutation operators illustrated with a running example. The results of the mutation analysis experiments were used to accomplish two objectives. The first one is for using mutation analysis in real cases, and make is feasible in practice. It consists in ranking the mutation operators to know the most important mutation operators that are necessary and sufficient to perform a good mutation analysis. This first study is a necessary step to apply mutation as a testing teschnique for security policies. The second objective is to empirically study the relation between security and functional tests and to compare between several criteria and strategies for security policy testing.

During the second part, we adapted the mutation analysis to evaluate SQL injection countermeasures. There are many approaches in the literature to deal with SQLIA. We started by presenting these approaches, their advantages and their drawbacks. Then, we proposed a strategy that is based on fault injection to evaluate these techniques. Faults are not injected in the security mechanism itself but to its environment (the application and the database). We consider a 3-tiers application. Our goal is to evaluate the robustness of the security mechanism. To do so, we inject faults in the behaviour of the application and we check whether the security mechanism resists or not to this perturbation. To each fault corresponds a mutation and we presented two types of mutants: general mutants that are used for all approaches and specific mutants that are dedicated to a given security mechanism.

With the mean of the “silver bullet” of mutation analysis, the report highlights the many issues a test expert has to deal with when facing the objective of testing a security policy and a security mechanism (such as SQLI countermeasures) for a real system. It shows that security policy correctness cannot be fully tested only with functional testing. Security appears as a specific testing objective with its test criteria and strategies. More fundamentally, the aspect of security mechanism testability in relation to system architecture appears as critical. Indeed, the way security mechanisms are spread over the system or centralized, the easiness or difficulty to relate a security rule to a piece of code are major issues to conduct the testing task. Future work may consider mutation as a way to compare various approaches to design a system for security (aspect weaving, traceability).

8. Glossary and acronyms

False positive
A wrong verdict which consists of rejecting something that should have been accepted.

False negative
A weak verdict which consists of accepting something that should have been rejected.

Mutant
A mutant is a version a program that contain one simple error (for example: a negation of a condition, plus instead of minus etc...).

Mutation operator
A mutation operator is used to inject a specific fault in the program and produce a mutant program containing this fault.

Mutation score
The mutation score (MS) computes the percentage of mutants killed by a test case: MS(t) = d/m, where d = dead mutants and m=all mutants (minus the equivalent ones).

OrBAC
Organization based access control is a security model. The main goal of OrBAC is to allow the policy designer to define a security policy independently of the implementation. The chosen method to fulfil this goal is the introduction of an abstract level. One important point in OrBAC is that each security policy is defined for and by an organization. Thus, the specification of the security policy is completely parameterized by the organization so that it is possible to handle simultaneously several security policies associated with different organizations.

Security flaw
An error of commission or omission in an information system (IS) that may allow protection mechanisms to be bypassed.

Security mechanism
A tool, piece of code or component (or combination of them) that protects an application from specific security attacks (like SQL injection etc…). It can be integrated in the application or be external to the application (for example controlling the data flow that goes to the application).

Security policy
A security policy defines the authorizations granted to users to access to the resources. A security policy that is based on OrBAC defines a set of security rules that can be « permission », « prohibition » or « obligation ». An example of security rule: *permission(Organization,Role,Activity,View,Context).*

SQL injection (SQLIA)
SQLIA is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is in fact an instance of a more general class of

vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.

Test case
A test case is a set of conditions or variables under which a tester will determine if a requirement or use case upon an application is partially or fully satisfied. In the report, we define a test case as a triplet: *intent*, input test sequence, oracle function. The *intent* may be either to test functional or security policy aspects

Test criterion
A test criterion specifies when a test cases set can be considered as adequate. A test criterion can be used to determine when a test set is sufficient for testing a piece of software, w.r.t. the effort and the expected quality of the resulting code. In practice, it is defined as a set of objects, taken from the specification or from the code, which must be exercised by the test set. In structural testing, executing each statement of the program under test is a basic test criterion. For security policy testing, covering each access control rule at least once is another basic criterion.

Oracle
The oracle function for a test case checks whether the result of the execution of the test case is correct or incorrect. In the case of mutation analysis, the oracle consists of comparing the result of the execution of the initial program with the mutant one.

Test strategy
A test strategy describes how the test cases are generated in order to reach a given test criterion. In the report, we compare an incremental test strategy, consisting of reusing functional test cases for SP testing, and independent test strategy, consisting in generating test cases from the OrBAC model.

Taint checking
It is a feature in some computer programming languages, such as Perl and Ruby, designed to increase security by preventing malicious users from executing commands on a host computer. Taint checks highlight specific security risks primarily associated with web sites which are attacked using techniques such as SQL injection or buffer overflow attack approaches.

Vulnerability
Weakness in a system allowing an attacker to violate the confidentiality, integrity, availability, access control, consistency or audit mechanisms of the system or the data and applications it hosts.

9. References

1. *CERT*, http://www.cert.org/stats/cert_stats.thml. [cited.
2. Thompson, H.H., *Why security testing is hard.* IEEE Security & Privacy Magazine, 2003. **1**(4): p. 83 - 86.
3. DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
4. Offutt, A.J., et al., *An Experimental Determination of Sufficient Mutant Operators.* ACM Transactions on Software Engineering and Methodology, 1996. **5**(2): p. 99 - 118.
5. Du, W. and A.P. Mathur. *Testing for Software Vulnerability Using Environment Perturbation*. in *International Conference on Dependable Systems and Networks*. 2000.
6. Ghosh, A., T. O'Connor, and G. McGraw, *An automated approach for identifying potential vulnerabilities in software*, in *IEEE Symposium on Security and Privacy*. 1998.
7. B. Miller, et al., *Fuzz revisited: A re-examination of the reliability of unix utilities and services*. 1995.
8. Lodderstedt, T., D. Basin, and J. Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002.
9. Briand, L. and Y. Labiche, *A UML-based approach to System Testing.* Software and Systems Modeling, 2002. **1**(1): p. 10 - 42.
10. Nebut, C., et al., *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
11. D. F. Ferraiolo, et al., *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224–274.
12. A. Abou El Kalam, et al., *Organization Based Access Control*, in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. 2003.
13. F. Cuppens, N. Cuppens-Boulahia, and M.B. Ghorbel. *High-level conflict management strategies in advanced access control models*. in *Workshop on Information and Computer Security (ICS'06)*. 2006.
14. Offutt, A.J., *Investigations of the software testing coupling effect.* ACM Transactions on Software Engineering and Methodology, 1992. **1**(1): p. 5 - 20.
15. Anley, C. *Advanced SQL Injection in SQL Server Applications*. [cited; Available from: http://www.nextgenss.com/reports/advanced sql injection.pdf.
16. Muthuprasanna M., Ke Wei, and Kothari, *Eliminating SQL Injection Attacks - A Transparent Defense Mechanism*, in *Web Site Evolution*. 2006.
17. S. W. Boyd and A.D. Keromytis. *SQLrand : Preventing SQL Injection Attacks* in *ACNS*. 2004.
18. William G. J. Halfond, Alessandro Orso, and P. Manolios. *Using positive tainting and syntax-aware evaluation to counter SQL injection attacks*. in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 2006.
19. F Valeur, D Mutz, and G. Vigna. *A Learning-Based Approach to the Detection of SQL Attacks*. in *Intrusion and Malware Detection and Vulnerability Assessment*. 2005.
20. YW Huang, et al. *Web application security assessment by fault injection and behavior monitoring*. in *International World Wide Web Conference*. 2003.
21. A.A, A., *An automated universal server level solution for SQL injection security flaw*, in *International Conference on Electrical, Electronic and Computer Engineering*. 2004